

南開大學

恶意代码分析与防治技术课程实验报告

实验四：IDA Pro、IDA Python 分析



学 院 网络空间安全学院
专 业 信息安全
学 号 2213041
姓 名 李雅帆
班 级 信安班

一、实验目的

1. 完成课本上的 Lab5 的实验内容，对 Lab05-01.dll 文件进行分析；
2. 在样本分析结果的基础上，编写对应的 Yara 检测规则；
3. 编写 IDA Python 脚本来辅助我们进行样本分析。
4. 熟练使用 IDA Pro 和 IDA Python。

实验原理

IDA Python 是基于 IDA Pro 的 Python 扩展，旨在通过编写脚本实现与 IDA Pro 的交互和自动化操作。它充分利用了 Python 的灵活性和强大标准库，为用户提供了一套完整的 API，用于操作和访问 IDA Pro 的各种功能和数据结构。通过这些 API，用户可以执行自动化任务，如识别函数、修改指令以及导入和导出数据等。

此外，IDA Python 支持事件驱动机制，允许用户注册和处理特定的 IDA 事件，例如载入二进制文件和分析完成。这使得用户能够编写回调函数，在特定事件发生时自动执行相应操作。

IDA Python 提供了一个交互式的 Python 解释器，用户可以在 IDA Pro 中直接编写和执行 Python 代码，实时进行数据查询和脚本运行。这些特性使得 IDA Python 成为恶意软件分析和反汇编的强大工具，大幅提高了分析效率和灵活性。

二、实验过程

• 准备工作

安装 IDA Pro 和 IDA Python，IDA 文件夹已经帮助我们配置好了 IDA Python 插件，接下来我们在控制台输入一些测试样例进行测试。

```

Python>print("I love NKU!")
I love NKU!
Python>a = 5
b = 3
print(a + b)
8
Python>name = "Alice"
greeting = "Hello, " + name + "!"
print(greeting)
Hello, Alice!

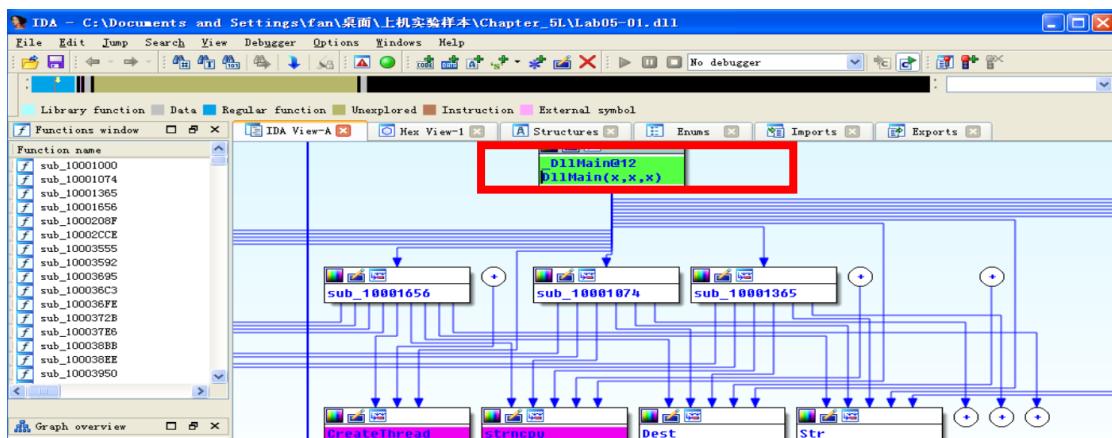
```

• Lab 05-1

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？

打开 IDA Pro，选择 Lab05-01.dll 打开，加载进去在图形化界面，程序自动定位到了 DLLMain 的地址。



右键选择 Text view，进入代码模式，就可以看到 DLLMain 的地址了。DLLMain 的地址就是在 .text 节的 0x1000D02E 处。

```

.text:1000D028    pop    esi
.text:1000D029    pop    ebx
.text:1000D02A    leave
.text:1000D02B    retm   8
.text:1000D02B ServiceMain endp
.text:1000D02B
.text:1000D02E
.text:1000D02E ; ----- S U B R O U T I N E -----
.text:1000D02E
.text:1000D02E ; BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD FdwReason, LPVOID lpvRes
.text:1000D02E D1lMain@12 proc near   ; CODE XREF: DllEntryPoint+4B1p
.text:1000D02E     ; DATA XREF: sub_100110FF+2D4o
.text:1000D02E     hinstDLL      = dword ptr  4
.text:1000D02E     FdwReason     = dword ptr  8
.text:1000D02E     lpvReserved   = dword ptr  0Ch
.text:1000D02E

```

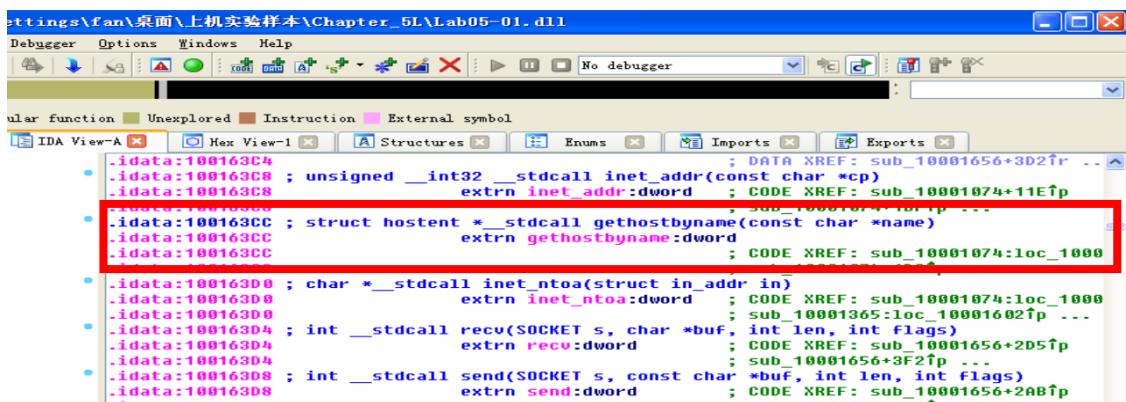
2. 使用 Imports 窗口并浏览到 gethostbyname, 导入函数定位到什么地址?

我们打开 Imports 窗口，然后搜索 gethostbyname。



双击该函数，就会跳转到反汇编窗口，定位到导入函数的地址。

gethostbyname 函数在.idata 节的 0x100163CC 处。



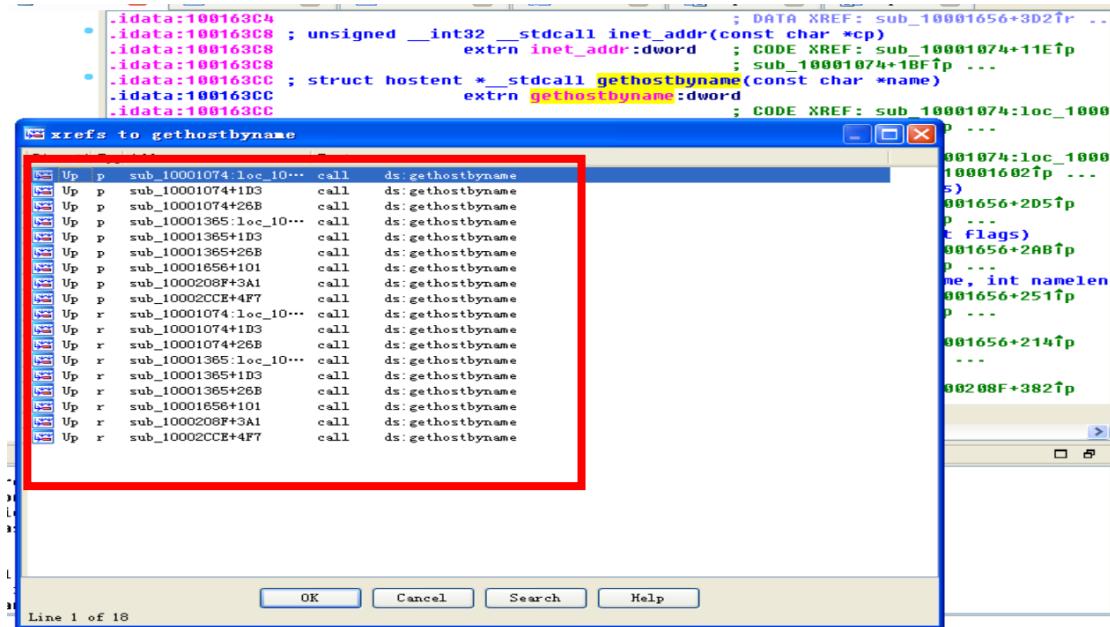
3. 有多少函数调用了 gethostbyname?

定位到 gethostbyname 函数位置，然后通过 Ctrl+X 快捷键定位其交叉引用情况。

类型 r 是被”读取”的引用，CPU 必须先读取这个导入项，再调用它；类型 p 是被调用的引用，这里显示了 18 行，但是并不是 18 个函数调用了这个 gethostbyname。（+后面的是偏移地址）

我们数一下发现，只有五个函数引用了 gethostbyname，每个 r 的类型，总有一个 p 的类型，所以被引用的次数是 9 次。

所以是五个函数调用了九次 gethostbyname。



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?

将外部引用窗口关闭, 然后按 G, 定位 0x10001757 跳转过去。

```
.text:10001748      jnz    loc_100017ED
.text:1000174E      mov    eax, off_10019040
.text:10001753      add    eax, 0Dh
.text:10001756      push   eax           ; name
.text:10001757      call   ds:gethostbyname
.text:1000175D      mov    esi, eax
.text:1000175F      cmp    esi, ebx
.jz    short loc_100017C8
```

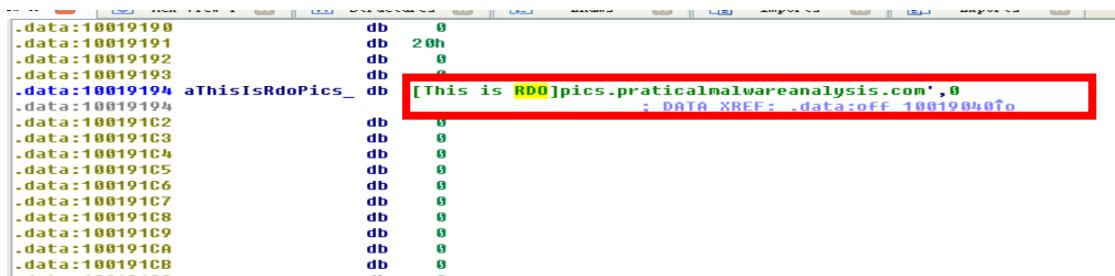
我们查看这块代码, 首先将 off_10019040 偏移地址位置的值赋值给了 eax, 然后给 eax 寄存器加上了一个 0Dh 的偏移量, 再通过 push 将 eax 的地址压入了栈中, 最后进行了 getpostbyname 的函数调用。

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

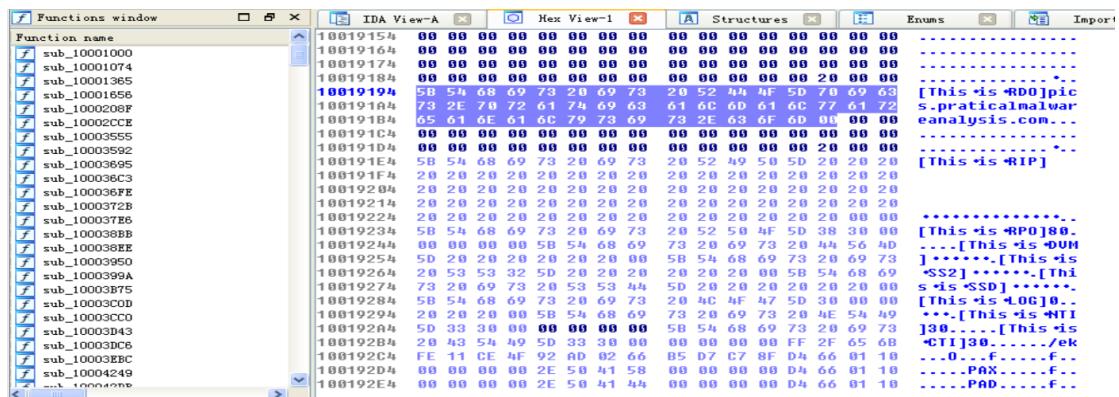
接下里我们跳到 off_1001940 定义的地方, 看到这个字符串: [This is RD
0]pics.praticalmalwareanalysis。



双击后面的 aThisIsRdoPics, 看到完整的字符串: [This is RDO]pics.pr
acticalmalwareanalysis.com。



接下来，我们通过打开“aThisIsRdoPics_”并使用十六进制窗口进行分析，我们发现前 13 个字节是 “[This is RDO]”。这表明，在接下来的代码中，地址 0x10001753 处的指令对寄存器 eax 执行了一个增加操作，使得 eax 的内容增加了 0Dh，这样就将 eax 中的值指向了 “practicalmalware-analysis.com” 的地址。然后，应用程序将 eax 的值压入栈中，作为后续函数调用“gethostbyname”的参数之一。



地址 0x10001757 处的代码对 “gethostbyname” 函数进行了调用，而该函数的参数指向了 “practicalmalwareanalysis.com” 的地址，因此可以推断出在这一位置发生的操作涉及到对 “practicalmalwareanalysis.com” 的访问或通信。

5. IDAPro 识别了在 0x10001656 处的子过程中的多少个局部那变量？

按 G，定位 0x10001656 跳转过去，然后会发现这些被 IDA 识别的变量。一共有 24 个参数，但是最后一个参数 dword ptr 4 是函数的第一个参数，而不是局部变量。

因此 IDA Pro 识别了在 0x10001656 处的子过程中的 23 个局部变量。

```

.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 -388h
.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. IDAPro 识别了在 0x10001656 处的子过程中的多少个参数?

IDA 识别的结果是传入了一个 LPVOID 类型的 lpThreadParameter，所以识别了一个参数。

```

.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 -388h
.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

```

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

按 SHIFT F12 调出 String 窗口, 查看\cmd. exe/c 字符串。

| Address | Length | Type | String |
|------------|----------|------|----------------|
| 0x10001656 | 0000000A | C | startxcmd |
| 0x10001656 | 0000000D | C | \\\cmd. exe /c |

双击\cmd.exe/c 字符串，可以看到\cmd.exe/c 字符串在 xdoors_d:10095B34 处。

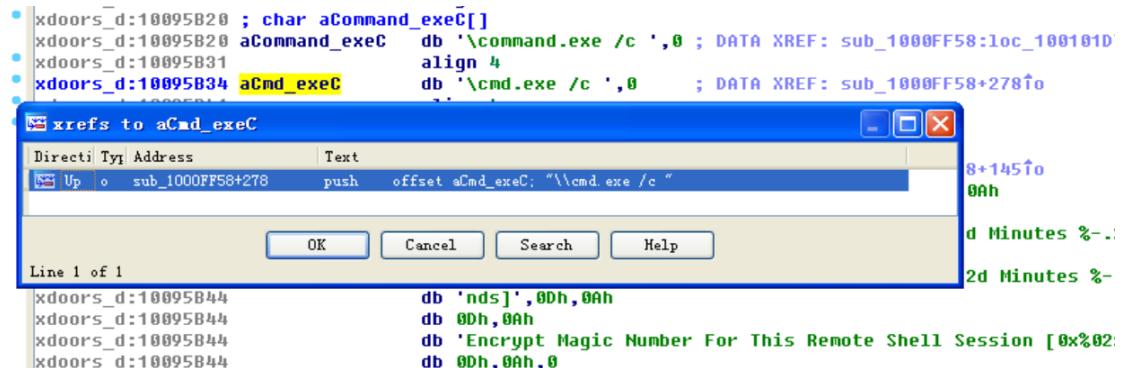
```

xdoors_d:10095B0C aCd          db 'cd',0           ; DATA XREF: sub_1000FF58+3AA↑o
xdoors_d:10095B0F align 10h
xdoors_d:10095B10 aExit        db 'exit',0         ; DATA XREF: sub_1000FF58+38D↑o
xdoors_d:10095B15 align 4
xdoors_d:10095B18 aQuit        db 'quit',0         ; DATA XREF: sub_1000FF58+36F↑o
xdoors_d:10095B1D align 10h
xdoors_d:10095B20 ; char aCommand_execC[]
xdoors_d:10095B20 aCommand_execC db '\command.exe /c ',0 ; DATA XREF: sub_1000FF58:loc_100101D7↑o
xdoors_d:10095B21 align 4
xdoors_d:10095B34 aCmd_execC  db '\cmd.exe /c ',0 ; DATA XREF: sub_1000FF58+278↑o
xdoors_d:10095B41 align 4
xdoors_d:10095B44 ; char aHiMasterDDDDDD[]
xdoors_d:10095B44 aHiMasterDDDDDD db 'Hi,Master [%d/%d/%d %d:%d:%d]',0Dh,0Ah
xdoors_d:10095B44           ; DATA XREF: sub_1000FF58+145↑o
xdoors_d:10095B44 db 'WellCome Back...Are You Enjoying Today?',0Dh,0Ah
xdoors_d:10095B44 db 0Dh,0Ah
xdoors_d:10095B44 db 'Machine Uptime [%-.2d Days %-.2d Hours %-.2d Minutes %-.2d Secon'
xdoors_d:10095B44 db 'ds]',0Dh,0Ah
xdoors_d:10095B44 db 'Machine IdleTime [%-.2d Days %-.2d Hours %-.2d Minutes %-.2d Seco'
xdoors_d:10095B44 db 'nds]',0Dh,0Ah
xdoors_d:10095B44 db 0Dh,0Ah
xdoors_d:10095B44 db 'Encrypt Magic Number For This Remote Shell Session [0x%02x]',0Dh,0Ah
xdoors_d:10095B44 db 0Dh,0Ah,0

```

8. 在引用\cmd.exe/c 的代码所在的区域发生了什么？

我们右键查找 aCmd_execC 引用。



然后跳到这个引用的地方，在进入该引用区域后，我们可以注意到一个函数位于内存地址 0x100101D0，并且该函数被推入到栈中。这一行为表明了在程序执行过程中的某一时刻，该函数将被调用。

```

.text:100101B0    lea    eax, [ebp+CommandLine]
.text:100101B6    mov    [ebp+StartupInfo.wShowWindow], bx
.text:100101BA    push   eax           ; lpBuffer
.text:100101BB    mov    [ebp+StartupInfo.dwFlags], 101h
.text:100101C2    call   ds:GetSystemDirectoryA
.text:100101C8    cmp    dword_1000E5C4, ebx
.text:100101CE    iz    short loc_100101D7
.text:100101D0    push   offset aCmd_execC; "\\cmd.exe /c"
.text:100101D5    jmp    short loc_100101DC
.text:100101D7    ;
.text:100101D7 loc_100101D7:           ; CODE XREF: sub_1000FF58+276↑j
.text:100101D7    push   offset aCommand_execC; "\\command.exe /c"
.text:100101DC loc_100101DC:           ; CODE XREF: sub_1000FF58+27D↑j
.text:100101DC    lea    eax, [ebp+CommandLine]
.text:100101E2    push   eax           ; Dest

```

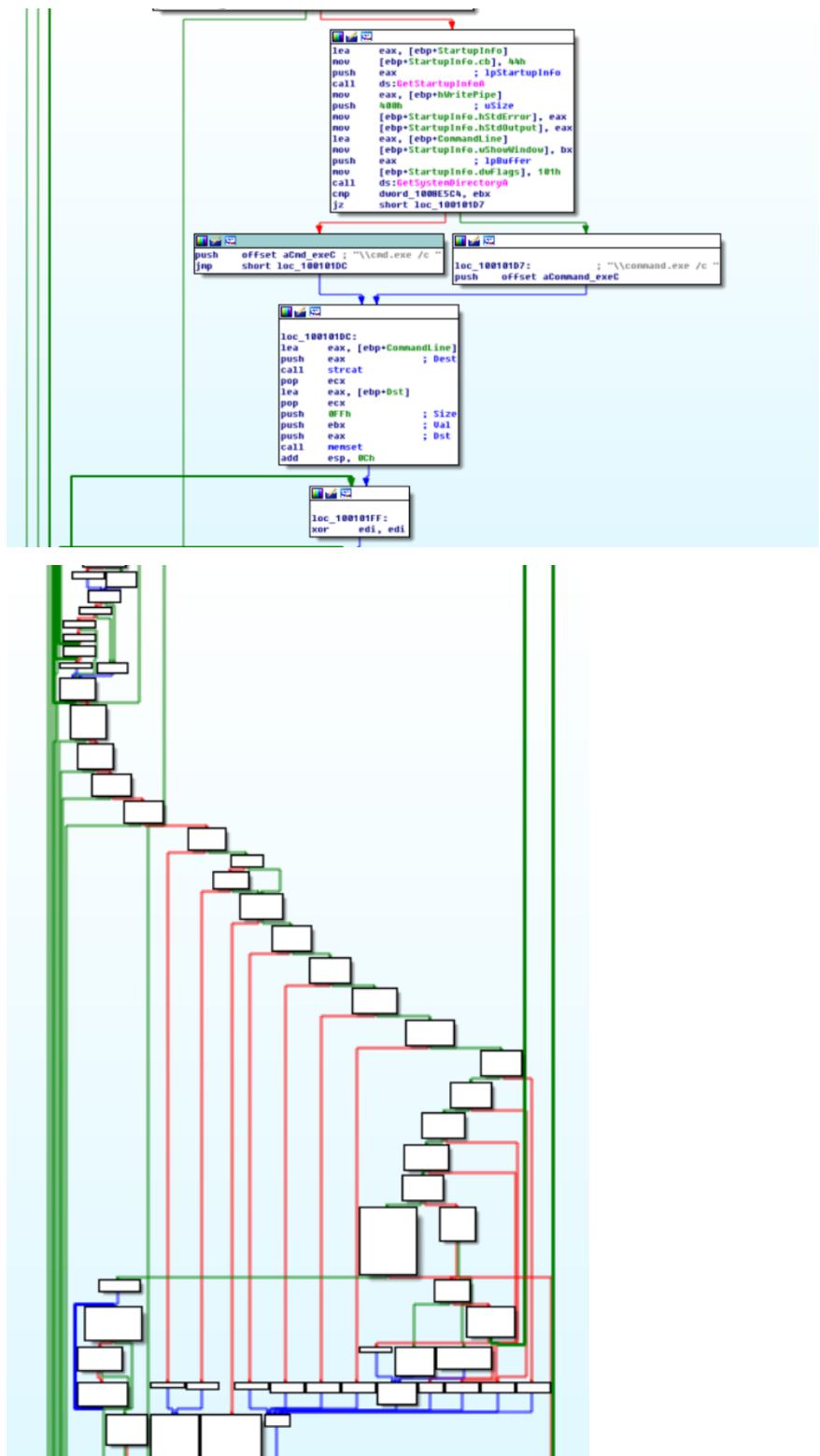
然而，在距离该字符串引用 0x1001009D 的位置，我们发现了一个字符串“aHiMasterDDDDDD”。通过进一步的调查，我们可以推断这个字符串是与程序中某种远程 shell 会话相关的信息。

```

.text:10010092
.text:10010096
.text:10010097
.text:1001009D
.text:100100A2
.text:100100A3
.text:100100A9
.text:100100AC
.text:100100AE

        movzx  eax, [ebp+SystemTime.wYear]
        push   eax
        lea    eax, [ebp+Dest]
        push   offset aHiMasterDDDDDD ; "Hi,Master [%d/%d/%d %d:%d:%d]\r\nWelCom"...
        push   eax
        call   ds:sprintf
        add    esp, 44h
        xor    ebx, ebx
        lea    eax, [ebp+Dest]

```



然后点开之后会发现这些有关于系统信息和系统操作的字符串，这是一个远程 shell 会话函数。更具体地说，根据被选中区域的字符串和上下文，我们可以合理地推测这里的代码片段可能是为了攻击者创建或启用一个远程 shell 会话。

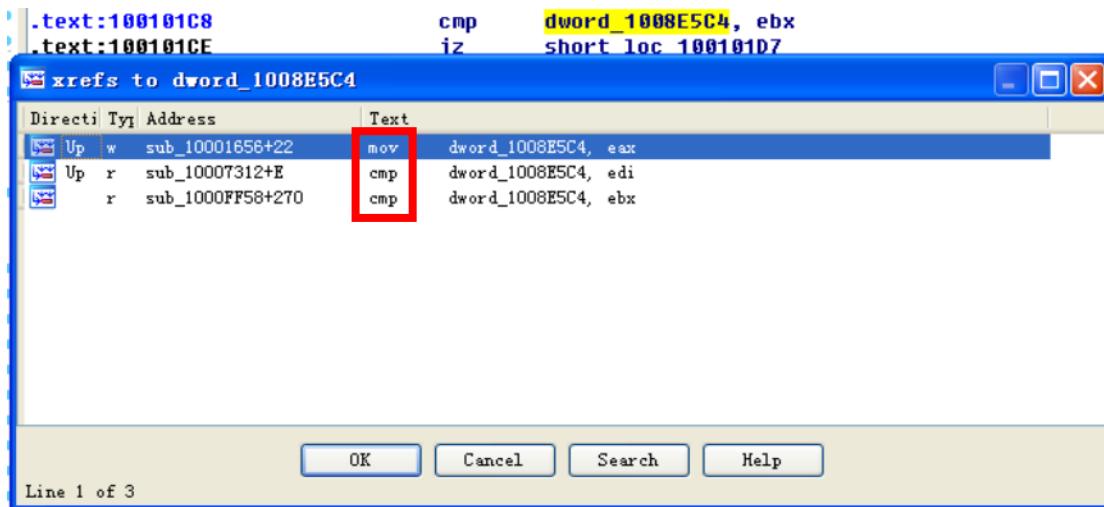
```
.text:100103FC ; -----
.text:100103FC
.text:100103FC loc_100103FC:
.text:100103FC     push    6          ; CODE XREF: sub_1000FF58+494↑j
.text:100103FC     lea     eax, [ebp+Dst]   ; Size
.text:100103FE     push    offset aUptime  ; "uptime"
.text:10010404     push    eax
.text:10010409     call    memcmp
.text:1001040F     add    esp, 0Ch
.text:10010412     test   eax, eax
.text:10010414     jnz    short loc_10010420
.text:10010416     push    [ebp+s]      ; s
.text:10010419     call    sub_10004DCA
.text:1001041E     jmp    short loc_100103F6
.text:10010420 ;
.text:10010420 loc_10010420:
.text:10010420     push    8          ; CODE XREF: sub_1000FF58+4BC↑j
.text:10010420     lea     eax, [ebp+Dst]   ; Size
.text:10010422     push    offset aLanguage ; "language"
.text:1001042D     push    eax
.text:1001042E     call    memcmp
.text:10010433     add    esp, 0Ch
.text:10010436     test   eax, eax
.text:10010438     jnz    short loc_10010444
.text:1001043A     push    [ebp+s]      ; s
.text:1001043D     call    sub_10004E79
.text:10010442     jmp    short loc_100103F6
.text:10010444 ;
.text:10010444 loc_10010444:
.text:10010444     push    9          ; CODE XREF: sub_1000FF58+4E0↑j
.text:10010444     lea     eax, [ebp+Dst]   ; Size
.text:10010446     push    offset aRobotwork ; "robotwork"
.text:1001044C     push    eax
.text:10010451     push    offset aRobotwork ; "robotwork"
.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
```

9. 在同样的区域，在 0x100101C8 处，看起来好像 dword_16008E5C4 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_16008E5C4 的呢？(提示：使用 dword_16008E5C4 的交叉引用。)

按 G 跳转到 0x100101C8，定位到函数 dword_1008E5C4。

| | | |
|----------------------|------------|-------------------------------------|
| .text:100101B6 | mov | [ebp+StartupInfo.wShowWindow], bx |
| .text:100101BA | push | eax ; lpBuffer |
| .text:100101BB | mov | [ebp+StartupInfo.dwFlags], 101h |
| .text:100101C2 | call | ds:GetSystemDirectoryA |
| text:100101C8 | cmp | dword_1008E5C4, ebx |
| .text:100101C8 | jz | short loc_100101D7 |
| .text:100101D0 | push | offset aCmd_execC ; "\\cmd.exe /c " |
| .text:100101D5 | jmp | short loc_100101DC |

查看与这个地址相关的交叉引用。结果显示，其他两处都是 cmp 函数，只有第一处是 mov 改变了它的值，也就是说只有一个引用是与写操作相关的。



双击跳到这个地方查看，我们发现了一条指令，该指令将 eax 寄存器的值赋给了 dword_1008E5C4。需要注意的是，eax 寄存器的值是前一条指令中某个函数调用的返回值。

```

.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
.text:10001687 mov    dword_1008E5C8, eax
.text:1000168C call   ds:Sleep
.text:10001692 call   sub_100110FF
.text:10001697 lea    eax, [esp+688h+WSAData]
.text:1000169E push   eax             ; lpWSAData
.push  202h             ; wVersionRequested

```

接下来我们查看子函数 sub_10003695。该函数包括一个对 GetVersionExA 函数的调用，用于获取当前操作系统版本的信息。接着，它会将 VersionInformation.dwPlatformId 与值 2 进行比较，确定 al 寄存器的值，即 eax 寄存器的值。

值 2 代表 WIN32_NT 系统，这段代码适用于判断当前 windows 的版本。获取的操作系统版本号会存储在 dword_1008E5C4 中，地址的值将是 0 或 1，反映了操作系统的类型。

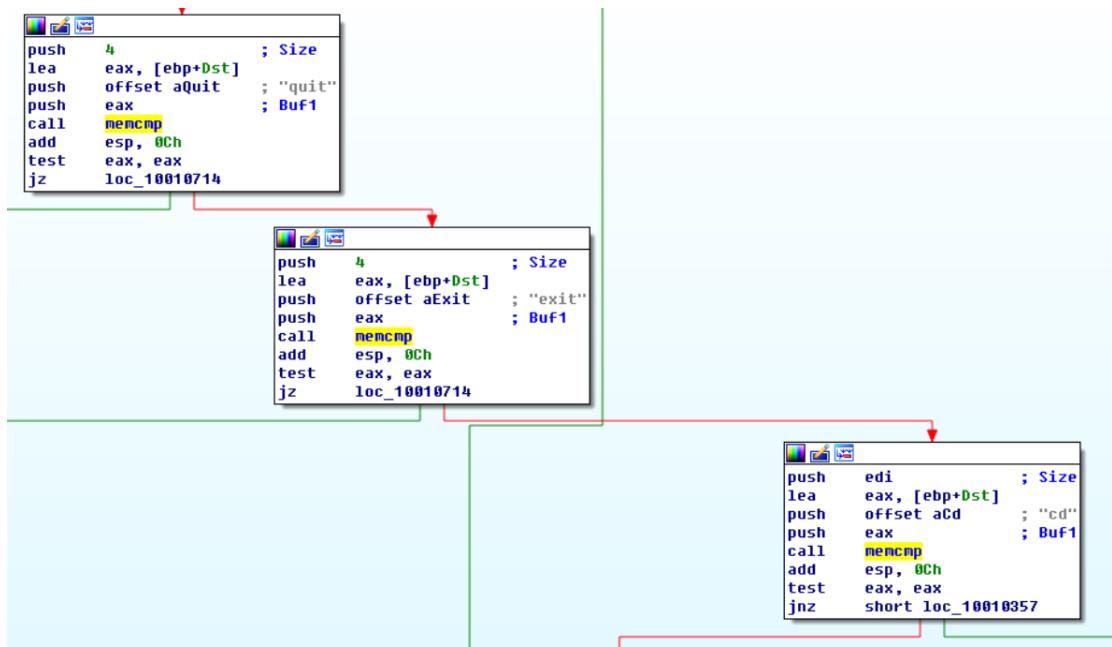
```

.text:10003695 sub_10003695    proc near                ; CODE XREF: sub_10001656+1D↑p
.text:10003695
.text:10003695 VersionInformation= _OSVERSIONINFOA ptr -94h
.text:10003695
.text:10003695     push    ebp
.text:10003695     mov     ebp, esp
.text:10003695     sub     esp, 94h
.text:10003695     lea     eax, [ebp+VersionInformation]
.text:10003695     mov     [ebp+VersionInformation.dwOSVersionInfoSize], 94h
.text:10003695     push    eax      ; lpVersionInformation
.text:10003695     call    ds:GetVersionExA
.text:10003695     xor     eax, eax
.text:10003695     cmp     [ebp+VersionInformation.dwPlatformId], 2
.text:10003695     setz   al
.text:10003695     leave
.text:10003695     retn
.text:10003695 sub_10003695 endp

```

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?

从 0x1000FF58 开始的函数, 第一个使用 memecp 如下图所示, 一开始是比较了 quit 和 eax 的值, 因为这两个值被压入了栈中。



找到 robotwork, 它首先压入了一个 robotwork 字符串指针, 然后压入了 `eax`, 然后 `call memcmp`, 如果两个数相同, 返回 0。

`test eax, eax`: 如果 `eax` 为 0, 则 ZF 置为 1, `jz` 跳转, `eax` 为 0 说明前面的 `memcmp` 比较的结果是相同, 也就是如果前面两个数相同, 则 `jz` 跳转, `jnz` 不跳转。

当字符串比较成功，memcmp 返回 0，jnz 不跳转，程序继续按从上到下的顺序执行，然后就是调用了 sub_100052A2 这个函数。

```

.text:10010444 ; -
.text:10010444
.text:10010444 loc_10010444:          push    9           ; CODE XREF: sub_1000FF58+4E0↑j
.text:10010444                           lea     eax, [ebp+Dst] ; Size
.text:10010446                           push    offset aRobotwork ; "robotwork"
.text:1001044C                           push    eax           ; Buf1
.text:10010451                           call    memcmp
.text:10010452                           add    esp, 0Ch
.text:10010457                           test   eax, eax
.text:1001045A                           jnz    short loc_10010468
.text:1001045C                           push    [ebp+1]      ; s
.text:10010461                           call    sub_100052A2
.text:10010466                           jmp    short loc_100103F6
.text:10010468 ; -
.text:10010468
.text:10010468 loc_10010468:          push    5           ; CODE XREF: sub_1000FF58+504↑j
.text:10010468                           lea     eax, [ebp+Dst] ; Size
.text:1001046A                           push    offset aMbase ; "mbase"
.text:10010470                           push    eax           ; Buf1
.text:10010475                           call    memcmp
.text:10010476                           add    esp, 0Ch
.text:1001047B                           test   eax, eax
.text:1001047E                           jnz    short loc_1001048F
.text:10010480                           push    [ebp+s]      ; s
.text:10010482                           call    sub_10004EE8
.text:10010485                           jmp    loc_100103F6
.text:1001048A ; -

```

查看 sub_100052A2 这个函数，该函数首先通过 push 压入 aSoftWareMicr os 处的值 SOFTWARE\Microsoft\Windows\CurrentVersion，然后通过调用 Reg OpenKeyxA 来读取该注册表的值。

13 / 30

```

.text:100052A2
.text:100052A3
.text:100052A5
.text:100052A8
.text:100052B2
.text:100052B3
.text:100052B8
.text:100052B9
.text:100052C0
.text:100052C7
.text:100052C9
.text:100052CB
.text:100052CC
.text:100052CE
.text:100052D0
.text:100052D1
.text:100052D7
.text:100052D9
.text:100052DB
.text:100052DC
.text:100052DF
.text:100052E0
.text:100052E5
.text:100052E7
.text:100052EC
.text:100052F1
.text:100052F7
.text:100052F9
.text:100052FB
.text:100052FE
.text:10005304
.text:10005309 ; -----



push    ebp
mov     ebp, esp
sub     esp, 60Ch
and     [ebp+Dest], 0
push    edi
mov     ecx, 0FFh
xor     eax, eax
lea     edi, [ebp+var_60B]
and     [ebp+Data], 0
rep stosd
stosw
stosb
push    7Fh
xor     eax, eax
pop     ecx
lea     edi, [ebp+var_20B]
rep stosd
stosw
stosb
lea     eax, [ebp+phkResult]
push    eax          ; phkResult
push    0F003Fh      ; samDesired
push    0             ; nOptions
push    offset aSoftwareMicros ; "SOFTWARE\Microsoft\Windows\CurrentVersion"
push    8000002h      ; hKey
call    ds:RegOpenKeyExA
test   eax, eax
jz     short loc_100053B9
push    [ebp+phkResult] ; hKey
call    ds:RegCloseKey
jmp    loc_100053F6

```

综上可知， robotwork 字符串比较成功， memcmp 返回 0， jnz 处判断语句就不会跳转，程序调用 call 指令进入 sub_100052A2 函数。

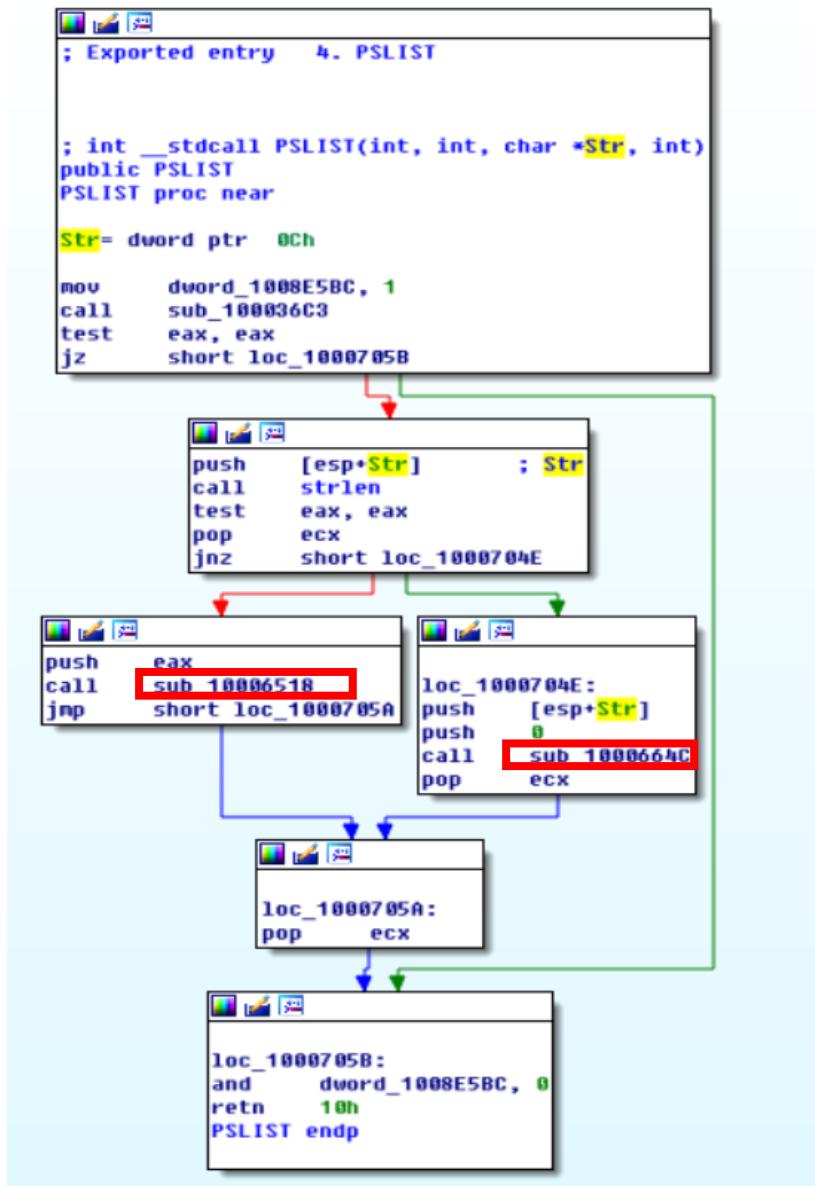
该函数首先用 RegOpenKeyExA 函数打开了注册表 SOFTWARE\Microsoft\Windows\CurrentVersion，然后查询键值，将这一信息返回给 push [ebp+s] 处传给该函数的网络 socket，然后将相关结构发送出去。

11. PSLIST 导出函数做了什么？

首先打开导出函数表，找到 PSLIST 函数。

| Name | Address | Ordinal |
|---------------|-------------------|---------|
| InstallRT | 0000000010000D847 | 1 |
| InstallSA | 0000000010000DEC1 | 2 |
| InstallSB | 0000000010000E892 | 3 |
| PSLIST | 00000000100007025 | 4 |
| ServiceMain | 000000001000CF30 | 5 |
| StartEXS | 0000000010007ECB | 6 |
| UninstallRT | 000000001000F405 | 7 |
| UninstallSA | 000000001000EA05 | 8 |
| UninstallSB | 000000001000F138 | 9 |
| DllEntryPoint | 0000000010001516D | |

双击进入函数，打开函数的图形化界面可以看出这个函数有两条执行大路径，然后执行的选择取决于这个 sub_100036C3，若其返回值为 1，则继续执行函数 sub_10006518 和 sub_1000664C。



我们首先关注 sub_100036C3 函数的内部逻辑，我们发现 sub_100036C3 函数根据一项条件进行判断，即判断操作系统是否为 Win32 且其版本是否大于 Windows 2000。若该条件成立，则 sub_100036C3 函数返回 1，否则返回 0。

```

; Attributes: bp-based frame
sub_100036C3 proc near
VersionInformation= _OSVERSIONINFOA ptr -94h
push    ebp
mov     ebp, esp
sub    esp, 94h
lea     eax, [ebp+VersionInformation]
mov     [ebp+VersionInformation.dwOSVersionInfoSize], 94h
push    eax      ; lpVersionInformation
call    ds:GetVersionExA
cmp     [ebp+VersionInformation.dwPlatformId], 2
jnz     short loc_100036FA

```



```

cmp     [ebp+VersionInformation.dwMajorVersion], 5
jb      short loc_100036FA

```

```

push    1
pop     eax
leave
retn

```

```

loc_100036FA:
xor    eax, eax
leave
retn
sub_100036C3 endp

```

接下来，我们深入研究 sub_10006518 和 sub_1000664C 两个函数。这两个函数均调用了 CreateToolhelp32Snapshot 函数和 sub_1000620C 函数。CreateToolhelp32Snapshot 函数的主要功能是获取主机系统的进程信息。

| | |
|---|--|
| <pre> push 2 ; dwFlags stosb call CreateToolhelp32Snapshot mov esi, ds:CloseHandle cmp eax, 0FFFFFFFh mov [ebp+hSnapshot], eax jz loc_10006640 </pre> | |
|---|--|

| | |
|---|--|
| <pre> push offset aProcessidProce ; "\r\n\r\nProcessID ProcessName ... call sub_1000620C pop ecx </pre> | |
|---|--|

```
rep stosd  
push    2          ; dwFlags  
call    CreateToolhelp32Snapshot  
cmp     eax, 0FFFFFFFh  
mov     [ebp+hSnapshot], eax  
jnz    short loc_100066DF
```

```
push    edi          ; Format  
call    sub_1000620C  
pop    ecx
```

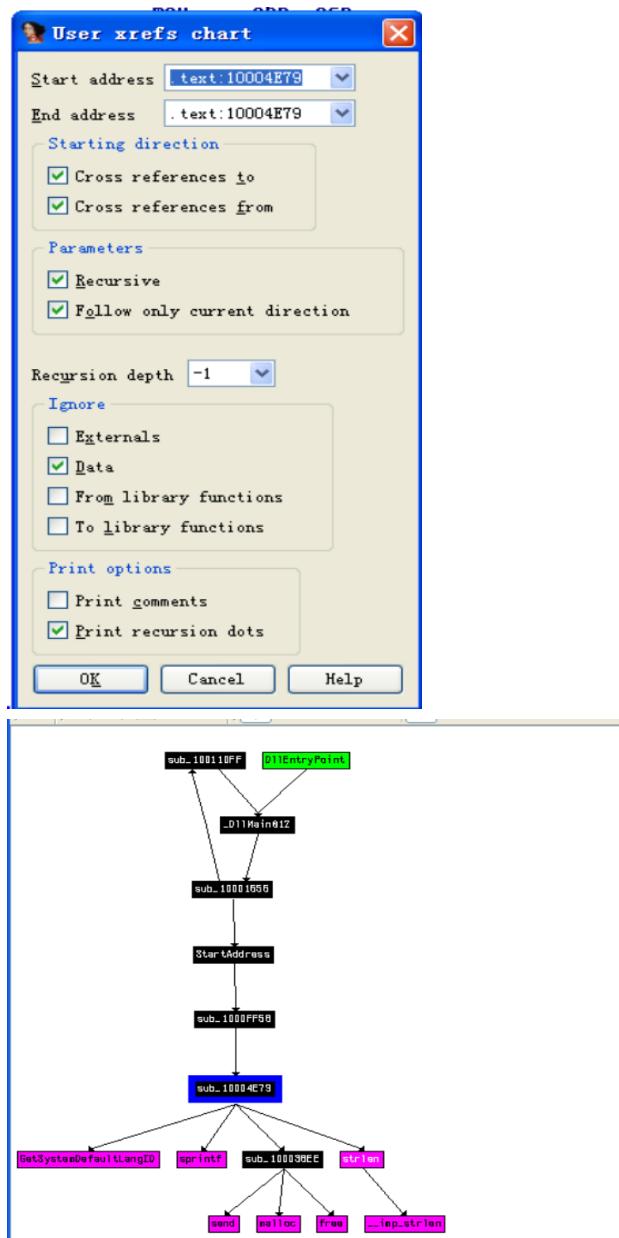
进一步研究 sub_1000620C 函数的内部逻辑，我们发现 sub_1000620C 函数的主要任务是将查询到的进程信息写入到一个文件中。

```
.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:1000620E         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:10006258         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
```

综上可知，PSLIST 导出功能将进程清单通过网络发送或在清单中查找特定进程名称，并获取有关其信息。

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？

View——Graphs——User Xrefs Chart，查看函数的交叉引用图。



函数 sub_10004E79 主要调用的有 GetSystemDefaultLangID 和 sprintf 和 sub_100038EE 和 strlen，而 sub_100038EE 主要调用了 send 和 malloc 和 free 和 __imp_strlen，然后 GetSystemDefaultLangID 是获取系统的默认语言的函数，send 是 socket 发送的函数。

由此我们将这个函数重命名为 send_languageID。

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

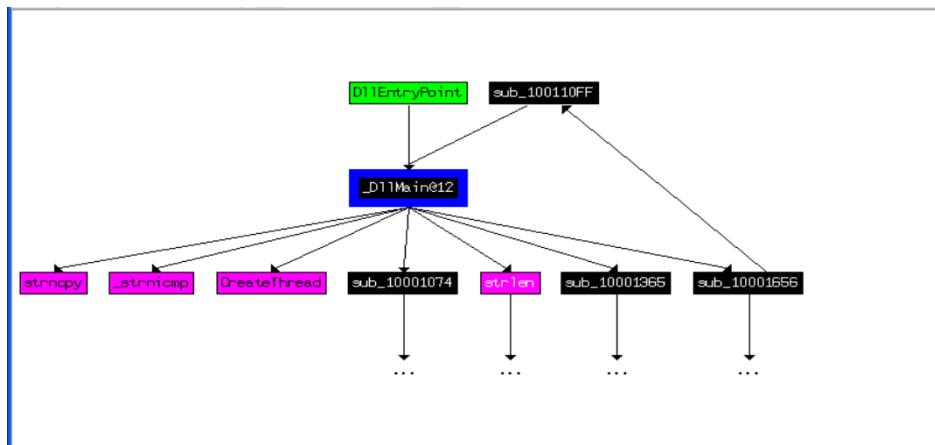
搜索 DllMain 找到这个函数的位置。

```
.text:1000002E hinstDLL      = dword ptr 4
.text:1000002E FdwReason     = dword ptr 8
.text:1000002E lpvReserved   = dword ptr 0Ch
.text:1000002E
.text:1000002E             mov    eax, [esp+FdwReason]
.text:10000032             dec    eax
.text:10000032             jnz    loc_10000107
.text:10000039             mov    eax, [esp+hinstDLL]
.text:1000003D             push   ebx
.text:1000003E             mov    ds:hModule, eax
.text:10000043             mov    eax, offset _10019044
.text:10000048             push   esi
.text:10000049             add    eax, 0Dh
.text:1000004C             push   edi
.text:1000004D             push   eax      ; Str
.text:1000004E             call   strlen
.text:10000053             mov    ebx, ds>CreateThread
.text:10000059             mov    esi, ds:_strnicmp
.text:1000005F             xor    edi, edi
.text:10000061             pop    ecx
.text:10000062             test   eax, eax
.text:10000064             jz    short loc_10000089
.text:10000066             mov    eax, offset _10019044
.text:10000068             push   7          ; MaxCount
.text:1000006D             add    eax, 0Dh
.text:10000070             push   offset aHttp_1 ; "http:///"
.text:10000075             push   eax      ; Str1
.text:10000076             call   esi : _strnicmp
.text:10000078             add    esp, 0Ch
.text:1000007B             test   eax, eax
.text:1000007D             jnz    short loc_10000089
.text:1000007F             push   edi
.text:10000080             push   edi
.text:10000081             push   edi
.text:10000082             push   offset sub_10001074
.text:10000087             jmp    short loc_100000BD
.text:10000089 ;-----
```

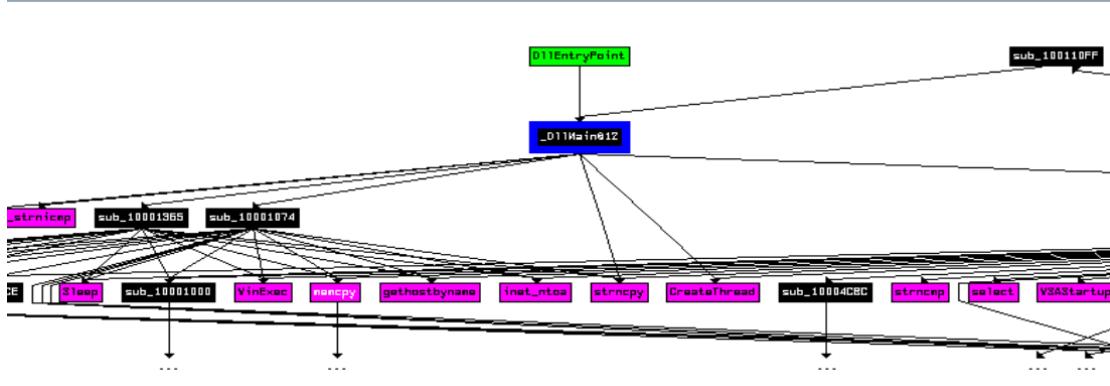
0000C42E 000000001000D02E: DllMain(x,x,x)

将 Recursion depth 改为 1, 可以看到调用深度为 1 的所有函数。

DllMain 在深度为 1 直接调用的 API 就是: strcpy、_strncpy、CreateThread、strlen。

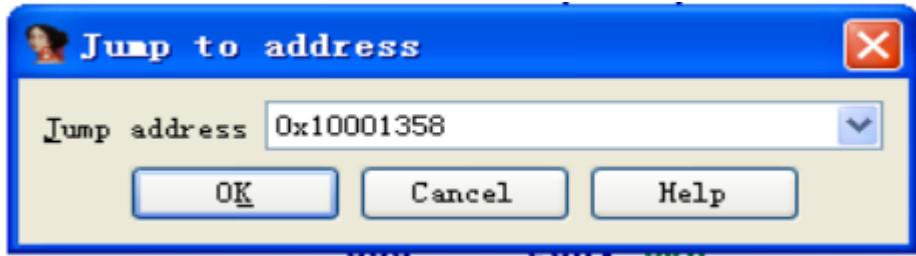


同样的方法查看深度为 2 时候的调用，调用关系十分复杂，我们可以看到非常多的 API 函数。



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, 这个程序会睡眠民多久?

先跳转到 0x10001358 处, 找到指定区域。



在调用 Sleep 函数之前, 将寄存器 eax 的值压入堆栈, 作为该函数的参数, 表示延迟时间。因此, 我们需要追溯到赋值给 eax 的地方以确定延迟的具体时长。在此之前, 程序将 off_10019020 的地址赋值给了 eax。

```
.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
.text:10001360
.text:10001365
```

导航到 off_10019020, 在该地址, 我们观察到了 [This is CTI]30 的字符串。

```
.data:10019020 off_10019020      dd offset aThisIsCti30 ; DATA XREF: sub_10001074:loc_10001341↑r
.data:10019020
.data:10019020
```

进一步导航到 aThisIsCti30, 我们确认其定义的字符串为 [This is CTI]30。

```
.data:10019298 aThisIsNti30      db '[This is NTI]30',0 ; DATA XREF: .data:off_10019024↑o
.data:100192A8
.data:100192A9
.data:100192AA
.data:100192AB
.data:100192AC aThisIsCti30      db '[This is CTI]30',0 ; DATA XREF: .data:off_10019020↑o
.align 10h
```

调用 Sleep 函数的汇编代码中，寄存器 eax 的值增加了 0DH（即 13 字节），这导致它现在指向 [This is CTI] 字符串。紧接着，eax 被压入堆栈并调用了 atoi 函数，这将其值转换为整数。因此，eax 现在的值为 30。随后，imul 指令将 eax 的值乘以 3E8h（即 1000），得到结果 30000 毫秒或 30 秒。

这意味着 Sleep 函数将导致程序暂停执行 30 秒。

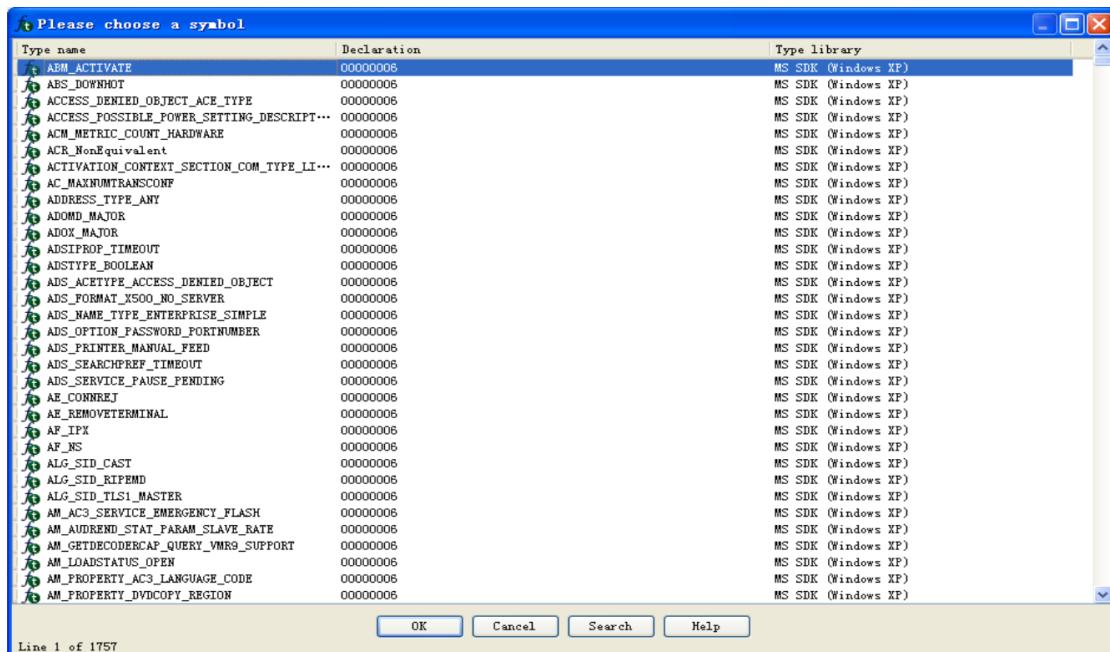
15. 在 0x10001701 处是一个对 socket 的调用。它的 3 个参数是什么？

定位到该地址处，发现 socket 函数被调用，这个函数有三个参数，分别为：af、type 和 protocol。通过查找交叉引用的函数，找到了调用它的函数地址，读出对应的三个参数的值：6、1 和 2。

```
.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, 0FFFFFFFh
.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 IDAPro 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？

在进行对指定的三个参数执行右键单击并选择“use standard symbolic constant”操作时，IDA Pro 将显示其为该特定值检索到的所有相应常量。



在所分析的代码中，已经向栈推送了三个数值，具体为 6、1 和 2。随后的注释提到了 protocol、type 以及 af。

结合先前关于 Python 中 socket 编程的经验，我们可以直接查询此函数，以确定需要输入的参数信息。通过搜索引擎查询，我们得知：type 代表传输模式或套接字类型；protocol 指代所采用的传输协议；而 af 表示地址族，即指定 IP 版本，如 IPv4 或 IPv6。

为了进一步明确这些参数中具体数字的含义，我们查阅了 Windows 平台的相关手册资料，从中得知：

af:指定地址族。2 指的是 AF_INET。用来设置 IPv4 socket 的值；

type: 指定套接字的类型。1 指的是 SOCK_STREAM；

protocol: 指定协议。6 指的是 IPPROTO_TCP 表示 TCP 协议，这个 socket 会被配置为基于 IPv4 的 TCP 连接在 HTTP 中。

应用了修改后参数如下：

The image shows three separate windows of the Microsoft Symbol Chooser tool, each displaying a table with Type name, Declaration, and Type library columns.

- The first window shows declarations for IPPROTO_TCP, MIB_IPPROTO_GGP, and IPPROTO_TCP. The declaration for IPPROTO_TCP is 00000006, and its type library is MS SDK (Windows XP). The declaration for MIB_IPPROTO_GGP is 00000005, and its type library is MS SDK (Windows XP). The declaration for IPPROTO_TCP is 00000006, and its type library is Visual C++ v6.
- The second window shows declarations for IRDA_PROTO_SOCK_STREAM, SOCK_STREAM, and SOCK_STREAM. The declaration for IRDA_PROTO_SOCK_STREAM is 00000001, and its type library is MS SDK (Windows XP). The declaration for SOCK_STREAM is 00000001, and its type library is MS SDK (Windows XP). The declaration for SOCK_STREAM is 00000001, and its type library is Visual C++ v6.
- The third window shows declarations for AF_INET and AF_INET. The declaration for AF_INET is 00000002, and its type library is MS SDK (Windows XP). The declaration for AF_INET is 00000002, and its type library is Visual C++ v6.

Below the windows, a portion of assembly code is shown:

```
.text:100016FB
.text:100016FD
.text:100016FF
.push    IPPROTO_TCP      ; protocol
.push    SOCK_STREAM       ; type
.push    AF_INET           ; af
.call    ds:socket
mov     edi, eax
cmp     edi, 0FFFFFFFh
jnz    short loc_10001722
call   ds:WSAGetLastError
```

17. 搜索 in 指令(opcode0xED)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 指令函数数的交叉引用，能发现进一步检测 VMware 的证据吗？

我们选择菜单中的 Search 中的 Sequence of Bytes 搜索 in 指令。



```

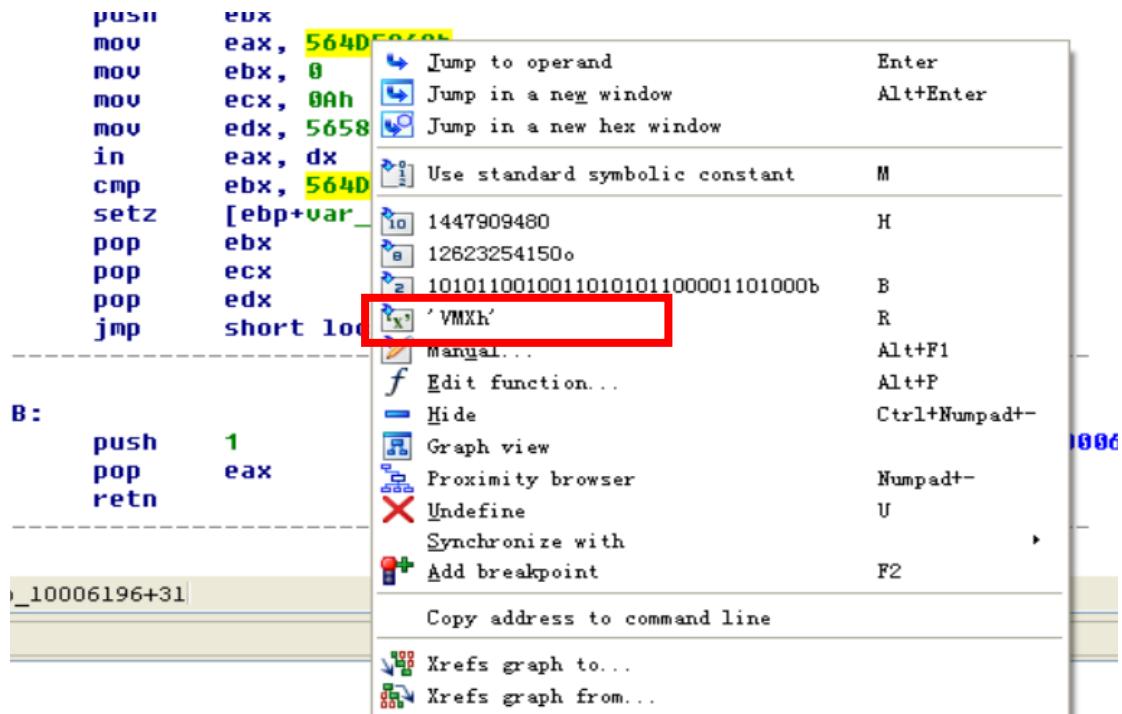
.text:10001098      sub_10001074      xor    ebp,   ebp
.text:10001181      sub_10001074      test   ebp,   ebp
.text:10001222      sub_10001074      test   ebp,   ebp
.text:100012BE      sub_10001074      test   ebp,   ebp
.text:1000135F      sub_10001074      xor    ebp,   ebp
.text:10001389      sub_10001365      xor    ebp,   ebp
.text:10001472      sub_10001365      test   ebp,   ebp
.text:10001513      sub_10001365      test   ebp,   ebp
.text:100015AF      sub_10001365      test   ebp,   ebp
.text:10001650      sub_10001365      xor    ebp,   ebp
.text:100030AF      sub_10002CCE      call   strcat
.text:10003DE2      sub_10003DC6      lea    edi,  [ebp+var_813]
.text:10004326      sub_100042DB      lea    edi,  [ebp+var_913]
.text:10004B15      sub_10004B01      lea    edi,  [ebp+var_213]
.text:10005305      sub_100052A2      jmp   loc_100053F6
.text:10005413      sub_100053F9      lea    edi,  [ebp+var_413]
.text:1000542A      sub_100053F9      lea    edi,  [ebp+var_213]
.text:10005B98      sub_10005B84      xor    ebp,   ebp
.text:100061DB      sub_10006196      in    eax,  dx
.text:10006305      sub_100062E9      lea    edi,  [ebp+var_1290]
.text:10006310      sub_100062E9      mov    [ebp+hModule], ebx
.text:10006318      sub_100062E9      call   ??2@YAPAXI@Z  ; operator new(uint)
.text:10006476      sub_100062E9      lea    ecx,  [ebp+hModule]
.text:100064A9      sub_100062E9      push   [ebp+hModule]  ; hModule
.text:1000671B      sub_1000664C      call   sub_1000620C
.text:10006C43      sub_10006BD5      jnz   short loc_10006C31
.text:10006D15      sub_10006CA7      jnz   short loc_10006D03
.text:10008687      sub_1000834E      call   sub_100073DA
.text:1000867A      sub_1000834E      call   sub_100073DA
.text:100086A7      sub_1000834E      call   sub_100073DA
.text:10008712      sub_1000834E      call   sub_10007442
.text:100096B1      sub_10009668      jnz   loc_100097A2
.text:10009A1A      sub_10009933      call   sub_100087A2
.text:1000A414      sub_1000A318      jmp   short loc_1000A402
.text:1000A501      sub_1000A4FD      xor    ebp,   ebp
.text:1000A5DD      sub_1000A4FD      xor    ebp,   ebp

```

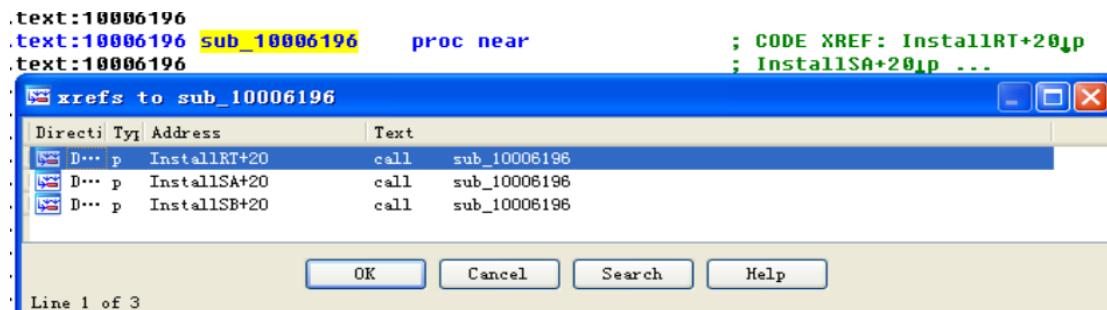
双击 sub_10006196 跳转到 in 指令地址处，发现了一些类似于字符串的十六进制数

| | | |
|------------------------|-----------|--------------------|
| .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 |
| .text:100061EB ; ----- | | |

右键，可以看到该地址对应字符串为“VMXh”。这表明相关代码部分应用了Vmware 的检测机制。



探索该指令相关的函数交叉引用。



细致地检查每个关联函数后，我们发现了”Found Virtual Machine”与”Install Cancel”这两个明确的标识。这些标识为Vmware 的进一步检测提供了确凿证据。

```
.text:10000870 loc_10000870:          ; CODE XREF: InstallIRT+1E↑j
.text:10000870    push    offset unk_1000E5F0 ; Format
.text:10000875    call    sub_10003592
.text:1000087A    mov     [esp+8+Format], offset aFoundVirtualMa ; "Found Virtual Machine,Install Cancel.."
.text:10000881    call    sub_10003592
.text:10000886    pop     ecx
.text:10000887    call    sub_10005567
.text:1000088C    jmp    short loc_100008A4
.text:1000088E ; 

.text:10000EEA loc_10000DEEA:        ; CODE XREF: InstallSA+1E↑j
.text:10000EEA    push    offset unk_1000E5F0 ; Format
.text:10000EEF    call    sub_10003592
.text:10000EF4    mov     [esp+8+Format], offset aFoundVirtualMa ; "Found Virtual Machine,Install Cancel.."
.text:10000FB     call    sub_10003592
.text:10000F00    pop     ecx
.text:10000F01    call    sub_10005567
.text:10000F06    jmp    short loc_1000DF1E
.text:10000F09 ; 

.text:10000EBB loc_10000E8BB:        ; CODE XREF: InstallSB+1E↑j
.text:10000EBB    push    offset unk_1000E5F0 ; Format
.text:10000EBC    call    sub_10003592
.text:10000ECC    mov     [esp+8+Format], offset aFoundVirtualMa ; "Found Virtual Machine,Install Cancel.."
.text:10000BD1    call    sub_10003592
.text:10000BD2    pop     ecx
.text:10000BD7    jmp    short loc_1000E8F4
```

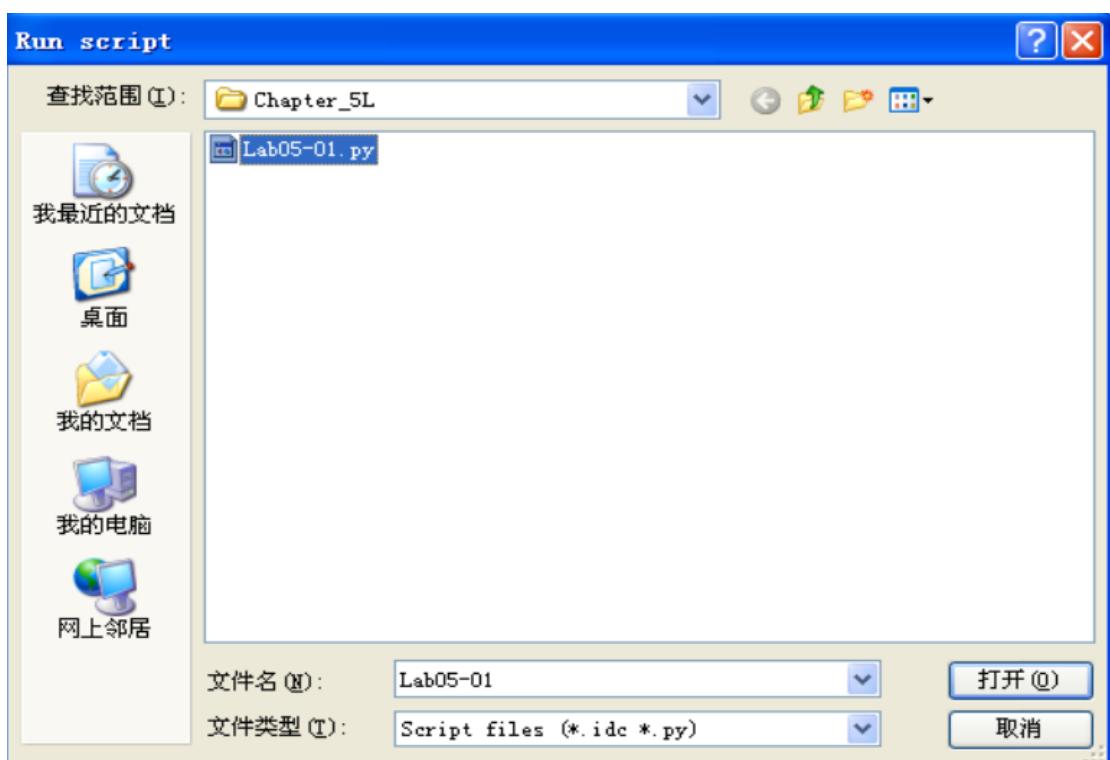
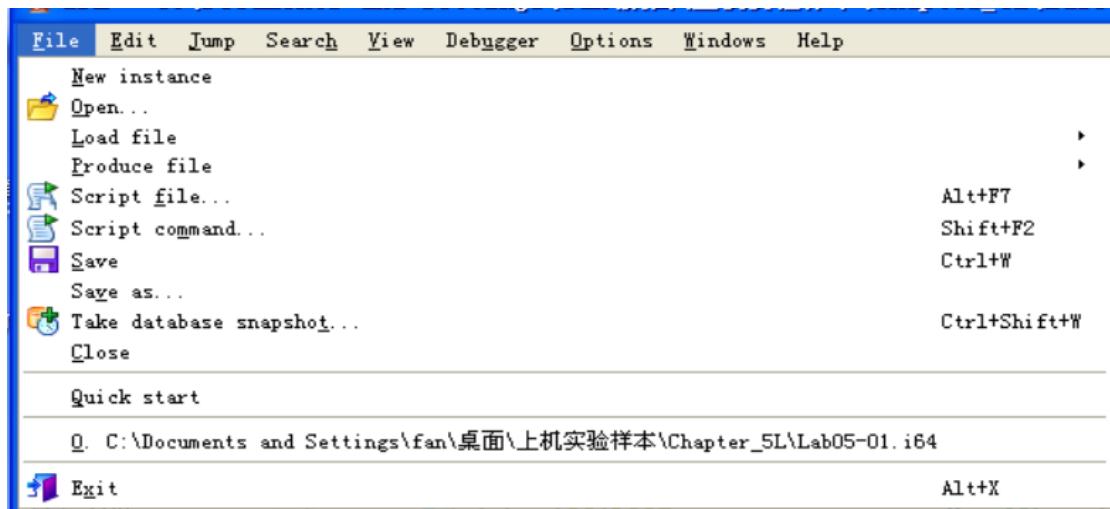
18. 将你的光标跳转到 0x1001D988 处, 你发现了什么?

观察到了一系列随机数据。

| | |
|----------------|--------|
| .data:1001D98B | db 3Ah |
| .data:1001D98C | db 27h |
| .data:1001D98D | db 75h |
| .data:1001D98E | db 3Ch |
| .data:1001D98F | db 26h |
| .data:1001D990 | db 75h |
| .data:1001D991 | db 21h |
| .data:1001D992 | db 3Dh |
| .data:1001D993 | db 3Ch |
| .data:1001D994 | db 26h |
| .data:1001D995 | db 75h |
| .data:1001D996 | db 37h |
| .data:1001D997 | db 34h |
| .data:1001D998 | db 36h |
| .data:1001D999 | db 3Eh |
| .data:1001D99A | db 31h |
| .data:1001D99B | db 3Ah |
| .data:1001D99C | db 3Ah |
| .data:1001D99D | db 27h |
| .data:1001D99E | db 79h |
| .data:1001D99F | db 75h |
| .data:1001D9A0 | db 26h |
| .data:1001D9A1 | db 21h |
| .data:1001D9A2 | db 27h |
| .data:1001D9A3 | db 3Ch |
| .data:1001D9A4 | db 38h |
| .data:1001D9A5 | db 32h |
| .data:1001D9A6 | db 75h |
| .data:1001D9A7 | db 31h |
| .data:1001D9A8 | db 38h |
| .data:1001D9A9 | db 36h |
| .data:1001D9AA | db 3Ah |
| .data:1001D9AB | db 31h |
| .data:1001D9AC | db 38h |
| .data:1001D9AD | db 31h |
| .data:1001D9AE | db 75h |
| .data:1001D9AF | db 33h |

19. 如果你安装了 IDAPython 插件(包括 IDAPro 的商业版本的插件#), 运行 Lab

05-01.py, 一个本书中随恶意代码提供的 IDAPython 脚本, (确定光标是在在 0x1001D988 处。)在你运行这个脚本后发生了什么?



```

.data:1001D988      db  78h ; x
.data:1001D989      db  64h ; d
.data:1001D98A      db  6Fh ; o
.data:1001D98B      db  6Fh ; o
.data:1001D98C      db  72h ; r
.data:1001D98D      db  20h
.data:1001D98E      db  69h ; i
.data:1001D98F      db  73h ; s
.data:1001D990      db  20h
.data:1001D991      db  74h ; t
.data:1001D992      db  68h ; h
.data:1001D993      db  69h ; i
.data:1001D994      db  73h ; s
.data:1001D995      db  20h
.data:1001D996      db  62h ; b
.data:1001D997      db  61h ; a
.data:1001D998      db  63h ; c
.data:1001D999      db  68h ; k
.data:1001D99A      db  64h ; d
.data:1001D99B      db  3Ah ; :
.data:1001D99C      db  3Ah ; :
.data:1001D99D      db  27h ; '
.data:1001D99E      db  79h ; y
.data:1001D99F      db  75h ; u
.data:1001D9A0      db  26h ; &
.data:1001D9A1      db  21h ; ?
.data:1001D9A2      db  27h ; '
.data:1001D9A3      db  3Ch ; <
.data:1001D9A4      db  3Bh ; ;
.data:1001D9A5      db  32h ; 2

```

20. 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 A, 就可以完成转化了, 多按几下, 就可以完成单一 ASCII 转化, 文本的内容为: 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

```

20. 使用一个文本编辑器打开这个脚本。它是如何工作的?

ScreenEA() 函数获取 IDA 调试窗口中光标所指向代码的地址, for 循环范围为连续 50 个字节, Byte 函数读取每个字节的值, 然后再将该值于 0x55 进行异或运算, 最后将结果输出返回到 IDA 对应的地址中。

```

Lab05-01 - 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)

```

• Yara 规则编写

```
rule lab5
{
strings:
    $string1 = "socket() GetLastError reports %d"
    $string2 = "WSAStartup() error: %d"
    $string3 = "HARDWARE\\DESCRIPTION\\System\\CentralProcessor\\0"
    $string4 = "SOFTWARE\\Microsoft\\Windows\\CurrentVersion"
    $string5 = "xkey.dll"

condition:
    filesize < 150KB and uint16(0) == 0x5A4D and uint16(uint16(0x3C)) == 0x00004550 and
all of them
}
```

运行结果如下：

```
PS C:\Users\lenovo> cd D:\fan\homework3\恶意代码分析与防治技术\计算机病
毒分析工具\yara
PS D:\fan\homework3\恶意代码分析与防治技术\计算机病毒分析工具\yara> .\ya
ra64.exe .\lab05-1.yara .\Lab05-01.dll
lab5 .\Lab05-01.dll
PS D:\fan\homework3\恶意代码分析与防治技术\计算机病毒分析工具\yara>
```

• 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。

```
import idautils

for func in idautils.Functions():
    flags = idc.GetFunctionFlags(func)

    if flags & FUNC_LIB or flags & FUNC_THUNK:
        continue

    dism_addr = list(idautils.FuncItems(func))
```

```

for line in disasm_addr:
    m = idc.GetMnem(line)

    if m == 'call' or m == 'jmp':
        op = idc.GetOpType(line, 0)

        if op == o_reg:
            print '0x%08x %s' % (line, idc.GetDisasm(line))

```

得到的输出如下：

The screenshot shows the IDA Pro interface with the assembly code output window open. The code listing consists of numerous calls to various Windows API functions, primarily dealing with file operations like RegOpenKeyExA, MoveFileExA, and CloseHandle, along with string manipulation functions such as sprintf and strcpy.

```

0x1000ec51 call    esi ; RegOpenKeyExA
0x1000ecdb call    esi ; strrchr
0x1000ed3e call    esi ; strrchr
0x1000edef call    edi ; RegSetValueExA
0x1000eeba call    edi ; sprintf
0x1000eecf call    edi ; sprintf
0x1000eea4 call    edi ; MovefileExA
0x1000ef58 call    edi ; MovefileExA
0x1000f0dc call    edi ; RegSetValueExA
0x1000f1b1 call    edi ; _strcmp
0x1000f1c3 call    edi ; _strcmp
0x1000f224 call    esi ; CloseHandle
0x1000f240 call    esi ; CloseHandle
0x1000f245 call    esi ; CloseHandle
0x1000f25f call    esi ; CloseHandle
0x1000f267 call    esi ; CloseHandle
0x1000f26c call    esi ; CloseHandle
0x1000f522 call    esi ; strncpy
0x1000f597 call    esi ; strncpy
0x1000f636 call    esi ; MovefileExA
0x1000f670 call    esi ; MovefileExA
0x1000f6ca call    esi ; SuspendThread
0x1000f6d6 call    esi ; SuspendThread
0x1000f7d7 call    esi ; sprintf
0x1000f89c call    esi ; sprintf
0x1000fa03 call    esi ; sprintf
0x1000fa30 call    esi ; strncpy
0x1000fa8a call    esi ; strncpy
0x1000fc2f call    esi ; sprintf
0x1000fc7e call    esi ; sprintf
0x1000fc96 call    esi ; sprintf
0x1000fd29 call    esi ; sprintf
0x1000fd82 call    esi ; sprintf
0x1000fd97 call    esi ; sprintf
0x1000ff27 call    esi ; sprintf
0x1000ff6e call    esi ; GetTickCount
0x1000ffc8 call    esi ; GetTickCount
0x10010feb call    esi ; fseek
0x10010fb0 call    esi ; fseek
0x1001109e call    esi ; fseek
0x100110b0 call    esi ; fseek
0x10011275 call    edi ; GetLastError
0x1001127b call    edi ; GetLastError

```

三、实验结论及心得体会

在进行“Lab 5-1”实验的过程中，我使用 Windows XP 下的 IDA Pro 工具，我们分析了 lab05-01.dll 程序，发现了它的多种特殊功能。

通过静态分析，我可以直接观察代码的结构和功能，识别关键的 API 调用和数据结构。这让我更清楚地了解恶意软件如何隐藏其行为，例如使用虚拟机检测和数据编码等技术。动态分析则让我看到恶意软件在实际运行中的目标，这为静态分析结果提供了重要的补充。

此外，IDAPython 脚本的编写让我体会到自动化逆向工程的强大。通过 IDAPython，我们可以快速定位特定的代码模式和函数调用，显著提高分析效率。这在处理复杂的二进制文件时尤其重要。

这次实验让我掌握了 IDA 的使用技巧，意识到持续学习的重要性。