



南開大學
Nankai University

网络空间安全学院
《恶意代码分析与防治技术》课程实验报告

实验四：IDA Pro 分析

姓名：王峥

学号：2211267

专业：信息安全

指导教师：王志、邓琮弋

2024 年 10 月 19 日

目录

| | |
|--------------------------|-----------|
| 1 实验目的 | 2 |
| 2 实验原理 | 2 |
| 3 实验过程 | 2 |
| 3.1 Lab 5-1 | 2 |
| 3.2 Yara 规则 | 16 |
| 3.3 IDA python | 18 |
| 4 实验结论及心得体会 | 19 |
| 4.1 实验结论 | 19 |
| 4.2 心得体会 | 20 |

1 实验目的

本次实验主要是使用 IDA Pro 和 IDA Python 工具对一个恶意 DLL 文件进行逆向工程分析。通过分析该 DLL 文件，学习如何查找入口函数、导入函数、函数调用、字符串和全局变量，以及如何分析恶意代码的功能和逻辑。有助于我们熟悉逆向工程的基本原理和工具，以及如何识别和理解恶意软件的行为。

2 实验原理

- **DllMain 函数地址:** DllMain 函数是动态链接库 (DLL) 的入口函数，它在 DLL 加载和卸载时被调用，其地址并不固定，因为每当 DLL 被加载到不同的进程地址空间时，它的实际内存地址都会有所不同。DllMain 函数的“地址”实际上是指它在 DLL 映像中的相对位置，这个位置在 DLL 被编译和链接时就已经确定了，我们需要找到它的地址，以便分析 DLL 的初始化和清理过程。
- **导入函数和 Imports 窗口:** 导入函数 (Imported Functions) 是外部库或 DLL 中供当前程序或模块调用的函数，它们通过特定的机制 (如动态链接) 被引入到程序中。Imports 窗口 (在编程环境或调试器中) 则是一个用于查看和管理当前程序或模块中所有导入函数的界面，它展示了程序与外部代码之间的依赖关系。
- **函数调用跟踪:** 函数调用跟踪是指记录和分析程序中函数调用的过程，包括调用顺序、参数传递、返回值处理等，以帮助开发者理解程序行为、调试错误或优化性能。
- **字符串定位:** 我们需要用 IDA Pro 的 Strings 窗口来定位特定字符串，如 `\cmd.exe/c`，并了解其在代码中的用途。
- **全局变量和局部变量分析:** 需要分析在代码中使用的全局变量和局部变量，以理解它们的作用。
- **API 函数调用分析:** API 函数调用分析是指对应用程序中 API (应用程序编程接口) 的调用过程进行追踪、记录和分析，旨在了解 API 的使用情况、性能表现、潜在问题以及用户行为等，需要查找和理解对 Windows API 函数的直接调用，包括 Sleep 和 socket，并分析其参数和功能。
- **指令分析:** 需要查找和分析特定指令，如 `in` 指令，以了解其在检测 VMware 环境中的使用。
- **IDA Python 脚本:** 使用 IDA Pro 的 Python 脚本功能来自动化一些分析任务，例如重命名函数或提取特定数据。

3 实验过程

3.1 Lab 5-1

分析在文件 Lab05-01.dll 中发现的恶意代码。

1. DllMain 的地址是什么?

为了查找 DllMain，我们要使用 IDA Pro 的 search 功能，即 **Search->Text** 进行搜索 DllMain。



图 3.1: DllMain 搜索结果

接下来我们跳转到 DllMain 即可找到其地址

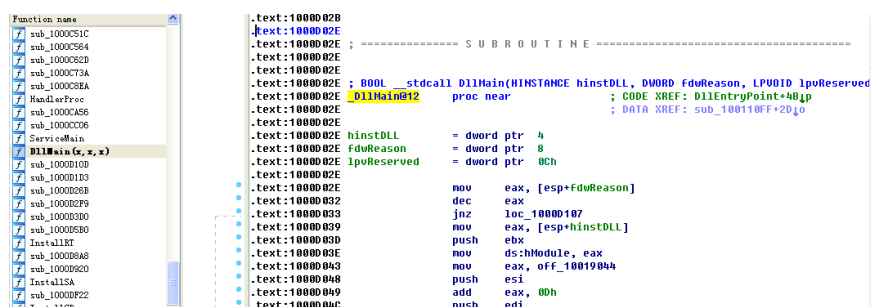


图 3.2: DllMain 地址

DllMain 的地址是 0x1000D02E, 在.text section

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

我们通过查看 Import 窗口, 即 View->Open subviews->Imports 查看 gethostbyname 导入函数, 并查看其地址

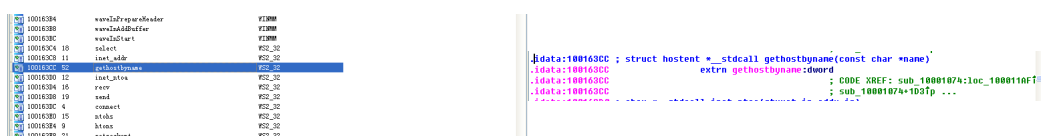


图 3.3: Process Monitor 查看

gethostbyname 的地址是 0x100163CC, 在.idata section.

3. 有多少函数调用了 gethostbyname?

我们使用 ctrl+x, 能查看 XREF 列表 (交叉引用列表), 上面有 18 行记录:

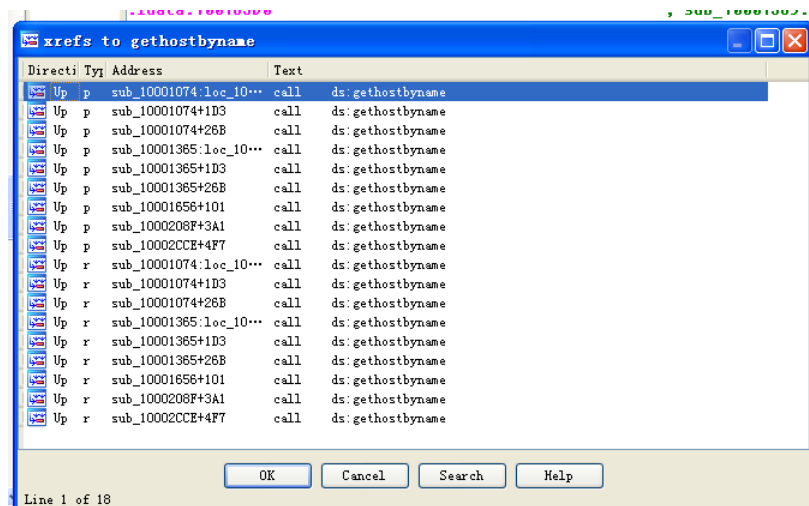


图 3.4: gethostbyname 的交叉引用列表

仔细观察表中，其中的 Type 中，p 代表调用，r 代表读。我们可以看到到：gethostbyname 被 5 个不同的函数调用 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？

在刚刚的交叉引用列表中，我们可以观察到 0x10001656+101 这个地址，我们双击跳转到 0x10001757 的地址上：



图 3.5: 0x10001757 地址

可以看到在 call 指令前还有一些指令，如 mov、add 和 push 指令等

- mov 指令：将 off_10019040 这个数放到 eax 中。
- add 指令：将其加上 0Dh 后，仍然放到 eax。
- push 指令：将 eax 推入栈中。

跳转到 off_10019040，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，指向了字符串 pics.practicalmalwareanalysis.com，我们可以猜测，这个网址就是 DNS 要解析的网址，所以解析该网址的 DNS 请求将会被触发。

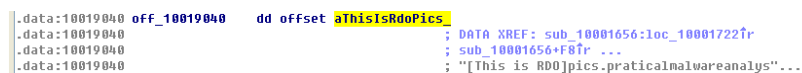


图 3.6: 查看 off_10019040

以上有许多参数, 举一个例子: `var_675 = byte ptr -675h`: 我们看到这定义了一个名为 `var_675` 的局部变量, 它是一个字节大小的变量, 位于基指针 `ebp` 的 `-675h` 偏移处。

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

等号右侧是负值的即为局部变量。

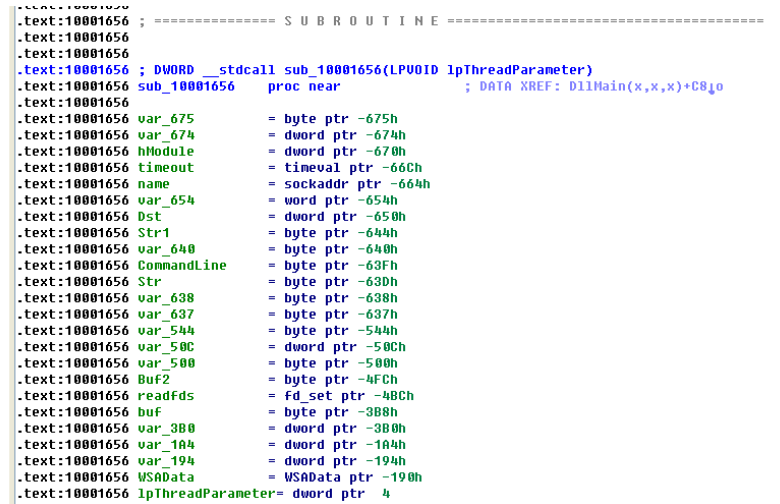


图 3.7: 局部变量

不难统计有 23 个局部变量不包含最后一行的 `lpThreadParameter`, 因为是传入的参数。

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

参数具有特征, 等号右侧是正值的即为参数, 因此上图中可以观察到传入的参数为 `lpThreadParameter`。所以 IDA Pro 识别了子过程中的 1 个参数。

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

首先我们通过 **View -> Open Subviews -> Strings** 直接进行搜索 `\cmd.exe /c`, 双击后可以看到该字符串位于 PE 文件 `xdoors_d` 节中的 `0x10095B34` 处, 在 `xdoors_d` 段。



图 3.8: 查找 `\cmd.exe /c` 字符串

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

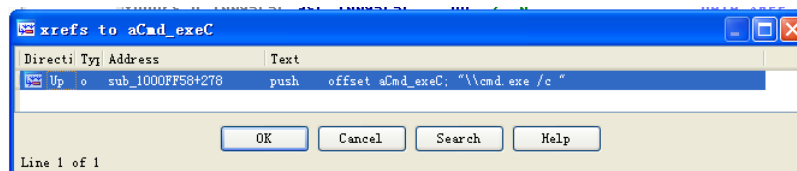


图 3.9: 交叉引用结果

8. 在引用 `\cmd.exe /c` 的代码所在的区域发生了什么?

在 `cmd.exe` 被交叉引用的地方, 即刚刚查看的 `text:100101D0` 处, 我们看到该命令被压栈。命令的后面还能看到用 `memcmp` 比较 `recv`、`quit`、`exit`、`cd`、`uptime` 等指令字符串。

```
.text:100101C8      cmp     dword_1008E5C4, ebx
.text:100101CE      jz      short loc_100101D7
.text:100101D0      push    offset aCmd_exeC ; "\\cmd.exe /c "
.text:100101D5      jmp     short loc_100101DC
```

图 3.10: 查看 sub_1000FF58+278 处地址

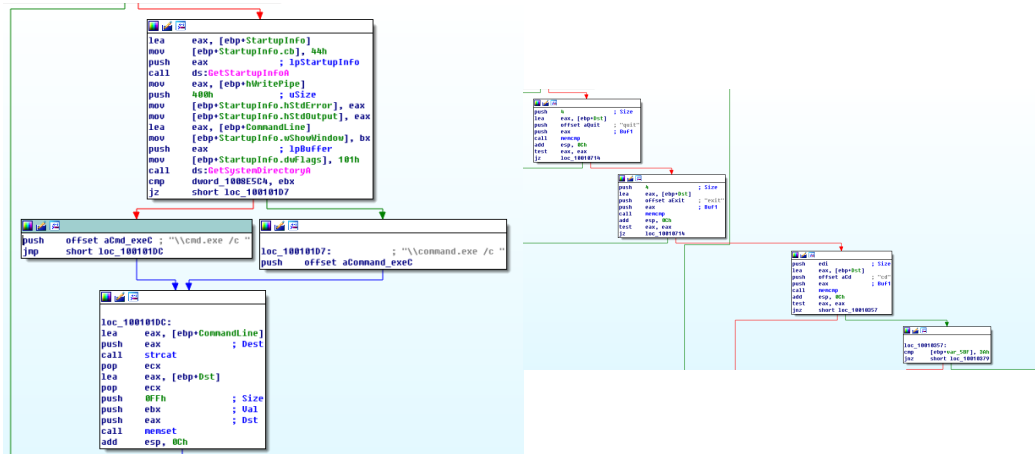


图 3.11: 部分代码查看

我们在在.text 0x1001009D 处有一个特殊的对某字符串的交叉引用。查看发现：

```
xdoors_d:10095844 ; char aHiMasterDDDDDD1
xdoors_d:10095844 aHiMasterDDDDDD db 'Hi,Master [%d/%d/%d %d:%d:%d]',00h,00h
xdoors_d:10095844 db 'Welcome Back...Are You Enjoying Today?',00h,00h
xdoors_d:10095844 db 'Machine UpTime [%d-%d Days %d-%d Hours %d-%d Minutes %d-%d Seconds]',00h,00h
xdoors_d:10095844 db 'Machine IdleTime [%d-%d Days %d-%d Hours %d-%d Minutes %d-%d Seconds]',00h,00h
xdoors_d:10095844 db 'Encrypt Magic Number For This Remote Shell Session [0x%02x]',00h,00h
xdoors_d:10095844 db 'WN,WAN,0'
xdoors_d:10095844 ; char asc_10095C5C
xdoors_d:10095844 asc_10095C5C db '>',0
xdoors_d:10095844 align 400h
xdoors_d:10095844 ends
```

图 3.12: aHiMasterDDDDDD 字符串内容

我们发现许多有趣的字符串：

- 'Hi,Master [%d/%d/%d %d:%d:%d]'：这个字符串似乎是用于显示或记录日期和时间的。‘%d’是一个格式说明符，用于表示整数。从上下文来看，像是一个欢迎消息，显示当前的日期和时间。
- 'Welcome Back...Are You Enjoying Today?'：这很像一个欢迎消息，在用户登录或启动某个应用程序时显示。
- 'Machine UpTime' 与 'Machine IdleTime'：这两个字符串分别用于显示或记录机器的运行时间，即从上次启动到现在的时间，以及显示或记录机器的空闲时间，即机器在没有用户活动的情况下的时间。
- 'Encrypt Magic Number For This Remote Shell Session [0x%02x]'：这个字符串似乎与远程 shell 会话的加密有关。它可能用于显示或记录用于加密的魔术数字。此外通过对 recv 和 send 的函数调用的了解，我们知道程序在进行内存非陪和发送数据的操作。因此，我们推测这个恶意代码与远程访问或远程 shell 有关，主要适用于远程控制或访问目标机器，当攻击者连接远程 shell 后他会像是这些信息，并提供靶机的信息与状态。

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 0x100101c8, 将 ebx 同全局变量 dword_1008E5C4 比较。

```
.text:100101B8      mov     [ebp+StartupInfo.dwFlags], 101h
.text:100101C2      call   ds:GetSystemDirectoryA
.text:100101C8      cmp     dword_1008E5C4, ebx
.text:100101CE      jz      short loc_100101D7
.text:100101D0      push   offset aCmd_exeC ; "\\cmd.exe /c "
.text:100101D5      jmp     short loc_100101DC
.text:100101D7      ;
```

图 3.13: 跳转至 100101c8 处

接下来双击跳到定义位置 (.data:1008E5C4), 再利用 Ctrl+X 查看交叉引用, 可以观察到它被引用了三次, 且 sub_10001656+22 处的语句修改了该值, 选中该位置。

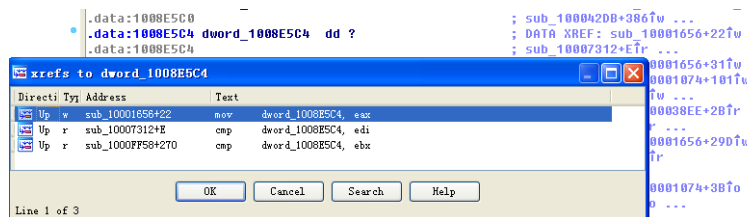


图 3.14: 查看.data:1008E5C4 的交叉引用

```
.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+68h+WSAData]
.text:1000169E      push   eax ; lpWSAData
.text:1000169F      push   202h ; wVersionRequested
.text:100016A4      call   ds:WSAStartup
```

图 3.15: 查看 10001678 处

我们可以看到 eax 是上一行调用的函数 sub_10003695 的返回值, 进入该函数:

```
.text:10003695 sub_10003695 proc near ; CODE XREF: sub_10001656+1D↑p
.text:10003695 ; sub_10003875+7↓p ...
.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
```

图 3.16: 查看函数

可以看到该函数进行如下操作:

- call ds:GetVersionExA: 调用了 GetVersionEx, 也即获取当前操作系统的信息。
- xor eax, eax: 将 eax 置 0。这是一个常见的汇编技巧, 用于快速清零寄存器。

- `cmp [ebp+VersionInformation.dwPlatformId],2`: 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, 通常情况下 `dwPlatformId` 的值为 2。
- `setz al`: 如果上一行的比较结果为真 (即 `dwPlatformId` 的值确实为 2), 则这行代码将 `al` 寄存器 (`eax` 的低 8 位) 的值设置为 1; 否则, 它将保持为 0。

综上, 该恶意代码判断当前操作系统是否 Windows 2000 或更高版本的 WindowsNT 系列, 如果是则将全局变量 `dword_1008E5C4` 的值置为 1。

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?

首先我们定位到目标地址 `0x1000FF58`, 之后观察到一系列 `memcmp`, 在 `0x10010452` 可以看到与 `robotwork` 的 `memcmp`。如果 `eax` 和 `robotwork` 相同, 则 `memcmp` 的结果为 0, 也即 `eax` 为 0。test 的作用和 `and` 类似, 只是不修改寄存器操作数, 只修改标志寄存器, 因此 `test eax,eax` 的含义是, 若 `eax` 为 0, 那么 `test` 的结果为 `ZF=1`。jnz 检验的标志位就是 `ZF`, 若 `ZF=1`, 则不会跳转, 继续向下执行, `call sub_100052A2`, 参数为 `socket` 类型。

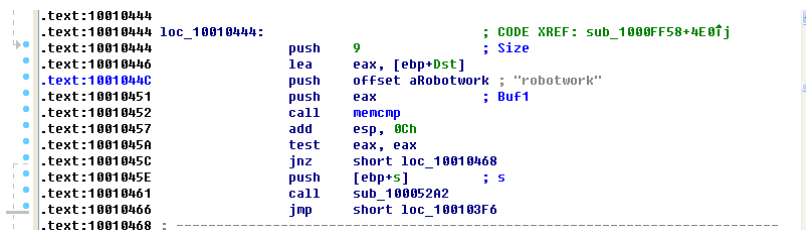


图 3.17: robotwork

接下来我们查看 `sub_100052A2` 的代码

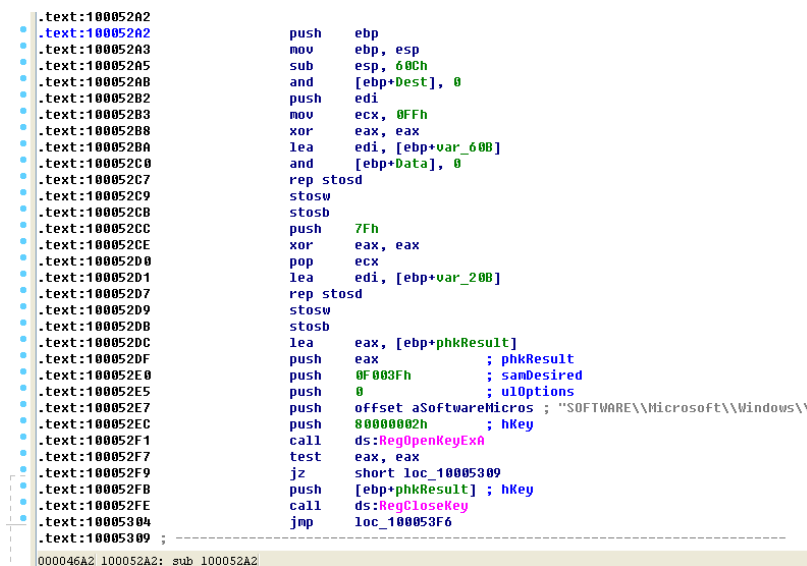


图 3.18: 查看 `sub_100052A2` 函数

可以找到 `0x100052E7` 处的 `aSoftWareMicros`, 其值为“SOFTWARE\Microsoft\Windows\CurrentVersion”。最后调用 `RegOpenKeyEx` 函数, 读取该注册表值。所以该恶意代码将注册表项 `HKLM\SOFTWARE\Microsoft\Wind` 的值会被查询, 并通过远程 shell 连接发送出去。

11.PSLIST 导出函数做了什么?

在 Exports 窗口中找到 PSLIST 查看：

| Name | Address | Ordinal |
|---------------|----------|---------|
| InstallRT | 1000D847 | 1 |
| InstallSA | 1000DEC1 | 2 |
| InstallSB | 1000E892 | 3 |
| PSLIST | 10007025 | 4 |
| ServiceMain | 1000CF30 | 5 |
| StartEXS | 10007ECB | 6 |
| UninstallRT | 1000F405 | 7 |
| UninstallSA | 1000EA05 | 8 |
| UninstallSB | 1000F138 | 9 |
| DllEntryPoint | 1001516D | |

图 3.19: 查看 Exports 导出表

双击查看：

```

.text:10007025 PSLIST      proc near                ; DATA XREF: .rdata:off_10017F78↓o
.text:10007025          = dword ptr 0Ch
.text:10007025 Str
.text:10007025          mov     dword_1000E5BC, 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

```

图 3.20: 查看 PSLIST 导出函数

看到该函数调用了 sub_100036C3, 双击查看：

```

.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:100036D8      push    eax                ; lpVersionInformation
.text:100036DD      call    ds:GetVersionExA
.text:100036E3      cmp     [ebp+VersionInformation.dwPlatformId], 2
.text:100036E9      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

```

图 3.21: 查看 sub_100036C3 函数

sub_100036C3 函数是在判断系统是否为 Win32 版本且版本是否大于 Win2000，如果符合则 return 1。然后我们进入 sub_10006518 和 sub_1000654 两个函数：

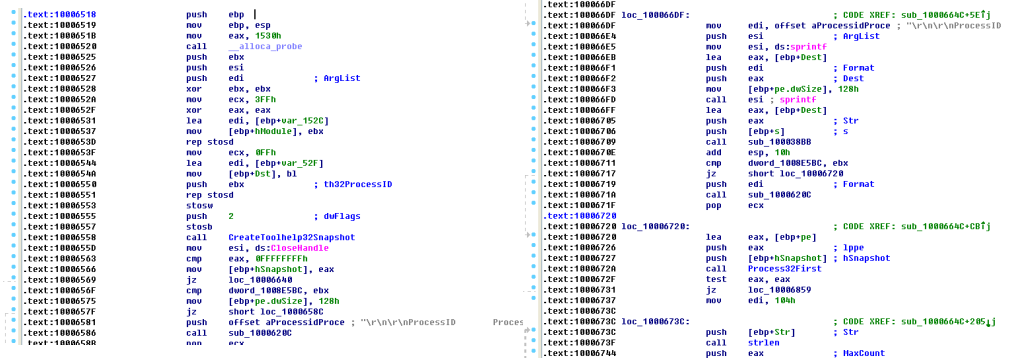


图 3.22: Process Monitor 查看

可以看到，两个函数都调用了 CreateToolhelp32Snapshot 函数和 sub_1000620C 函数，其中 CreateToolhelp32Snapshot 函数可以获取进程信息，sub_1000620C 函数可以将其查找的进程信息写入一个文件。所以 PSLIST 的作用就是获取进程信息，并将进程信息写入到文件之中形成一个列表。

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？

首先我们查找 sub_10004E79 函数的位置：



图 3.23: sub_10004E79 函数

(3) 使用图模式来绘制出对 sub_10004E79 的交叉引用图，View→Graphs→User xrefs chart:

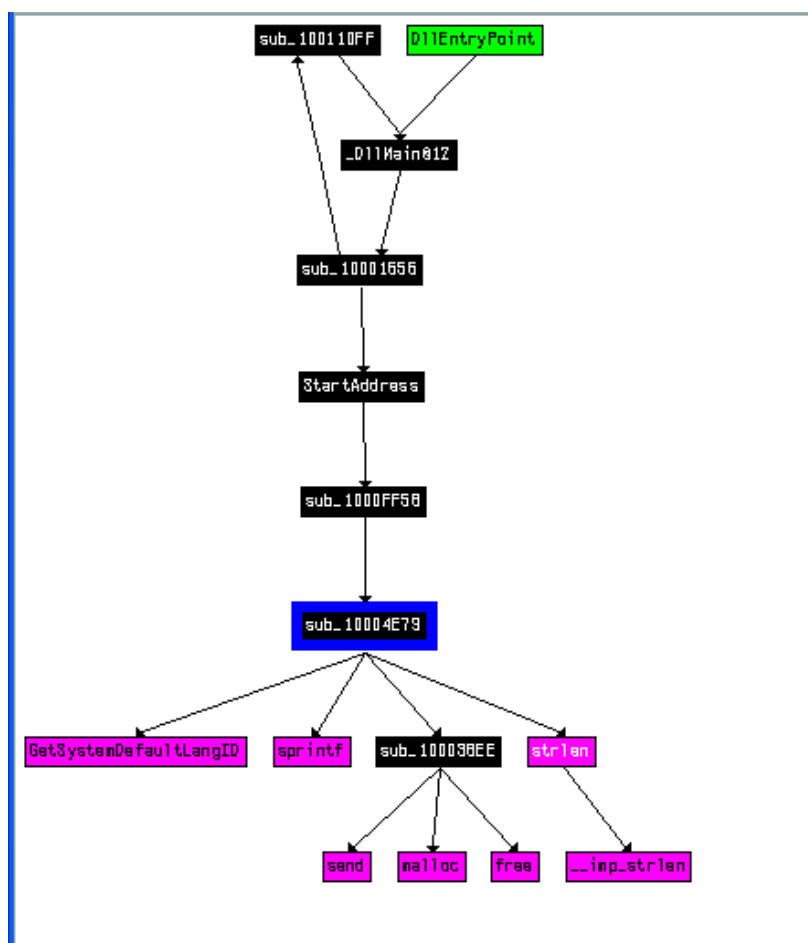


图 3.24: sub_10004E79 交叉引用图

我们可以观察到该函数调用 GetSystemDefaultLangID（获取系统默认 ID 对照表）并发送，该函数可能会通过网络 socket 发送语言标识符，这样就可以知道系统使用的是什么语言。所以我们通过在函数处点击右键 → Edit function，将其重命名为 “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？

首先跳转到 Dllmain 的起始位置 0x1000D02E，然后选择 **View -> Graphs -> User xrefs chart** 选项弹出对话框，曲线勾选 Cross references from，可以看到调用的 Windows API 为 CreateThread、strncpy、strlen 以及 _strnicmp。

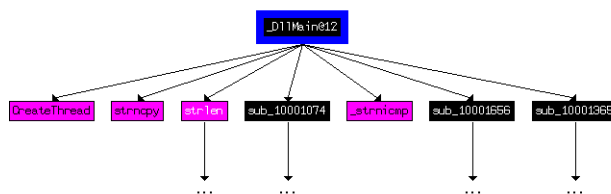


图 3.25: Dllmain 的交叉引用

我们将深度改为 2，查看：

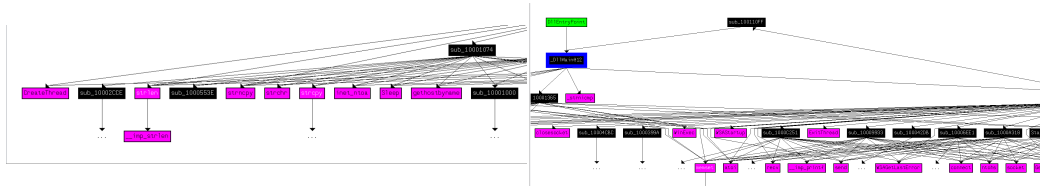


图 3.26: 深度为 2 调用部分查看

看到许多的 API 被交叉引用, 以及一些网络函数, 如如 socket, send, connect 等。

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?

```

.text:10001357      push     eax                ; dwMilliseconds
.text:10001358      call    ds:Sleep
.text:1000135E      xor     ebp, ebp
.text:10001360      jmp     loc_100010B4
.text:10001360      sub_10001074
.text:10001360      endp
.text:10001365      ; ===== SUBROUTINE =====
.text:10001365      ; DWORD __stdcall sub_10001365(LPVOID lpThreadParameter)
.text:10001365      sub_10001365      proc near                ; DATA XREF: DllMain(x,x,x)+8A40
.text:10001365      File             = FILE ptr -54h
.text:10001365      var_30           = word ptr -30h
.text:10001365      in               = in_addr ptr -2Ch
.text:10001365      Dst             = byte ptr -20h
.text:10001365      var_1F          = byte ptr -1Fh
.text:10001365      lpThreadParameter = dword ptr 4

```

图 3.27: 0x10001358 处代码

可以看到:

- 在 0x10001358 处有一个对 sleep 的调用, 并且 sleep 使用一个参数, 也就是要休眠的毫秒数 dwMilliseconds。这个参数被传入了 eax 中压入栈中供 sleep 使用。
- imul eax, 3E8h:eax 被乘了 0x3E8(十进制是 1000)。而 eax 正好是前面 atoi 函数调用的结果返回值。对 atoi 调用的结果被乘以 1000, 得到要休眠的毫秒数。
- mov eax, off_10019020: off 100192 被赋给 EAX。双击查看:

```

.data:10019020 off_10019020      dd offset aThisIsCti30 ; DATA XREF: sub_10001074:loc_10001341tr
                                ; sub_10001365:loc_10001632tr ...
                                ; "[This is Cti]30"

```

图 3.28: .data:10019020 查看

可以看到它指向了一个字符串 [This is Cti]30。再次回到代码位置, 其中 add eax, 0Dh: 0xD 被加到 EAX 上作为偏移。因此, 通过将 EAX 指向 30 来调用 atoi 函数, 将字符串" 30" 转换为数字 30。然后将 30 乘以 1000, 得到 30,000 毫秒 (即 30 秒), 这就是程序休眠的时间。睡眠的时间应为 $30 \times 1000 = 30000$ 毫秒, 也即 30 秒。

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

跳转后可以看到三个参数名: af、type、protocol, 分别是 6、1、2。

```

.text:100016FB      push     6                ; protocol
.text:100016FD      push     1                ; type
.text:100016FF      push     2                ; af
.text:10001701      call    ds:socket

```

图 3.29: 查看 socket 调用参数

```

.text:100061C6      push     ebx
.text:100061C7      mov     eax, 564D5868h
.text:100061C8      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

```

图 3.32: 双击查看结果

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?

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

```

.text:100061FB      push     IPPROTO_TCP
.text:100061FD      push     SOCK_STREAM
.text:100061FF      push     AF_INET
.text:10006170      call     ds:socket

```

图 3.30: 替换常量名

“Use Standard Symbolic Constant” 对话框列出了 IDA Pro 对特定值具有的所有常量。在本次实验的 Q15、Q16 中, 6 表示 IPPROTO_TCP; 1 表示 SOCK_STREAM; 2 表示 AF_INET。AF_INET 用于连接连接对象是 IPv4 时、SOCK_STREAM 用于连接方式使用 TCP 时候、IPPROTO_TCP 用于继续指明传输的方式是 TCP。所以此 socket 将配置为通过 Ipv4 的 TCP 协议。

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?

首先使用 Search→Sequence of Bytes, 并勾选 Find all occurrences, 搜索 in 指令, 如下所示:

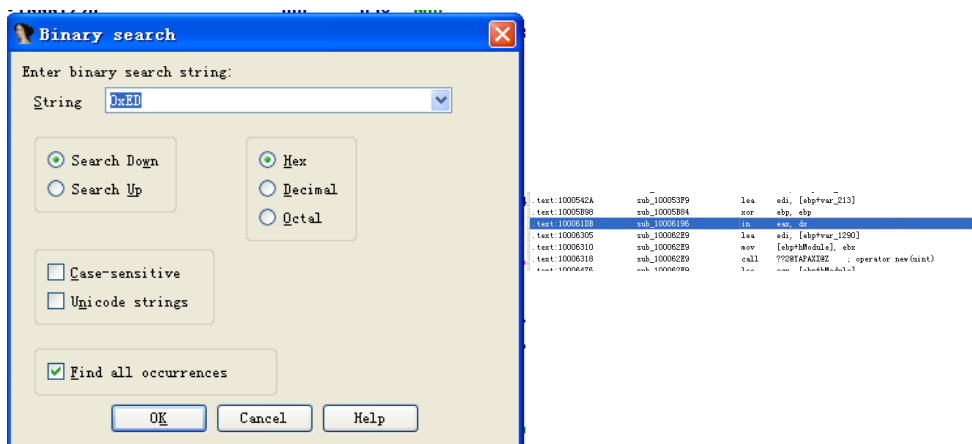


图 3.31: in 指令搜索

双击跟随后我们查看到:

将.text:100061DB 的十六进制串, 变为字符串查看。(键盘 R 键), 看到了 eax 中储存了字符串“VMXh”, 确认了这段代码采用了反虚拟机技巧。

```

.text:1000610H      in     eax, dx
.text:1000610C      cmp     ebx, 'UMXh'
.text:100061E2      setz    byte ptr [ebp-1Ch]
.text:100061E6      pop     ebx

```

图 3.33: 转变字符串查看

找到函数的入口处 sub_10006196, 打开交叉引用, 选择第一个, 可以看到该函数的后文中出现了字符串 “Found Virtual Machine,Install Cancel.”, 印证了猜测。

```

.text:1000867      call    sub_10006196
.text:100086C      test    al, al
.text:100086E      jz      short loc_100088E
.text:1000870      ; CODE XREF: InstallRT+1Efj
.text:1000870 loc_1000870:
.text:1000870      push    offset unk_1000E5F0 ; Format
.text:1000875      call    sub_1000E592
.text:100087A      mov     [esp+8+Format], offset aFoundVirtualMa ; "Found Virtual Machine,Install Cancel."
.text:1000881      call    sub_1000E592
.text:1000886      pop     ecx
.text:1000887      call    sub_1000E567
.text:100088C      jmp     short loc_1000884

```

图 3.34: 查看 sub_10006196

18. 将你的光标跳转到 0x1001D988 处, 你发现了什么?

目标地址位于 .data 节, 全部为一些十六进制的数据乱码, 我们无法获得信息, 所以我们可以结合 Python 脚本进行解密。

```

.data:1001D987      db      0
.data:1001D988      db      2Dh ; -
.data:1001D989      db      31h ; 1
.data:1001D98A      db      3Ah ; :
.data:1001D98B      db      3Ah ; :
.data:1001D98C      db      27h ; '
.data:1001D98D      db      75h ; u
.data:1001D98E      db      3Ch ; <
.data:1001D98F      db      26h ; &
.data:1001D990      db      75h ; u
.data:1001D991      db      21h ; ?
.data:1001D992      db      3Dh ; =
.data:1001D993      db      3Ch ; <
.data:1001D994      db      26h ; &
.data:1001D995      db      75h ; u
.data:1001D996      db      37h ; 7
.data:1001D997      db      34h ; 4
.data:1001D998      db      36h ; 6
.data:1001D999      db      3Eh ; >
.data:1001D99A      db      31h ; 1
.data:1001D99B      db      3Ah ; :
.data:1001D99C      db      3Ah ; :
.data:1001D99D      db      27h ; '
.data:1001D99E      db      79h ; y
.data:1001D99F      db      75h ; u
.data:1001D9A0      db      26h ; &
.data:1001D9A1      db      21h ; ?
.data:1001D9A2      db      27h ; '
.data:1001D9A3      db      3Ch ; <
.data:1001D9A4      db      38h ; ;
.data:1001D9A5      db      32h ; 2
.data:1001D9A6      db      75h ; u
.data:1001D9A7      db      31h ; 1

```

图 3.35: 查看 0x1001D988

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?

使用 File→Script file→引入 idc 文件, 打开之后效果如下:

```

.data:1001D987 db 0
.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 6Fh ; o
.data:1001D99C db 6Fh ; o
.data:1001D99D db 72h ; r
.data:1001D99E db 2Ch ; ,
.data:1001D99F db 20h
.data:1001D9A0 db 73h ; s
.data:1001D9A1 db 74h ; t
.data:1001D9A2 db 72h ; r
.data:1001D9A3 db 69h ; i
.data:1001D9A4 db 6Eh ; n
.data:1001D9A5 db 67h ; g
.data:1001D9A6 db 20h
.data:1001D9A7 db 64h ; d

```

图 3.36: 引入 idc 文件

可以看到之前 Q18 的十六进制乱码已经被 script file 解密，变为有序有意义的字符。

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？

可以使用键盘 A 键，使其连为一句话，如下所示：

```

.data:1001D987 db 0
.data:1001D988 aXdoorIsThisBac db 'xdoor is this backdoor, string decoded for Practical Malware Anal'
.data:1001D989 db 0
.data:1001D98A db 0
.data:1001D98B db 0
.data:1001D98C db 0
.data:1001D98D db 0

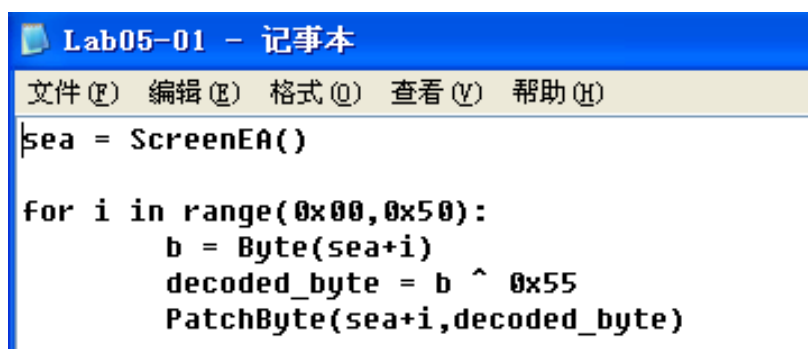
```

图 3.37: 连起来一句话

这句话连起来为 “xdoor is this backdoor, string decoded for practical Malware Analysis Lab :)1234”。

21. 使用一个文本编辑器打开这个脚本。它是如何工作的？

打开结果如下：



```
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)
```

图 3.38: 查看 Lab05-01.py 结果

- ScreenEA() 函数：用来获取光标所在位置
- for 循环：在 0x00 到 0x50 间循环（连续 50 个字节循环）
- Byte () 函数：读取每个字节的值，逐字相加
- decoded_byte = b 0x55：与 0x55 异或
- 最后进行输出，并将其返回到 IDA 对应的地址中。

3.2 Yara 规则

根据我们上述分析，结合使用 IDA 的字符串窗口的查看得出的字符串，编写对应的 Yara 规则重点字符串：

- \\cmd.exe /c：标志病毒调用远程 Shell 窗口会话使用的命令行。
- Hi,Master：最有标志性的字符串，代表着与远程 Shell 窗口交互连接时，对方对本方客户的问好。
- robotwork：与网络相关的字符串
- SOFTWARE...CurrentVersion：病毒对宿主机版本的判断。
- pics.practicalmalwareanalysis.com：之前 lab 中就出现过多次的网站。结合以上分析，编写规则如下：

Yara 规则

```
1 private global rule IsPE {
2     condition:
3         filesize < 150KB and
4         uint16(0) == 0x5A4D and // MZ 头
5         uint32(uint32(0x3C)) == 0x00004550 // PE 头
6 }
7
8 rule get {
9     strings:
10         $get = "gethostbyname"
```

```
11     $url = "pics.practicalmalwareanalysis.com"
12     $ver = "VersionInformation.dwPlatformId"
13     $version = "VersionInformation.dwMajorVersion"
14     $a = "malloc"
15     $b = "send"
16     $c = "free"
17     condition:
18         ($get or $url or $ver or $version or $a or $b or $c) and IsPE
19 }
20
21 rule remote_shell_session {
22     strings:
23         $remote = "Remote Shell Session"
24     condition:
25         $remote and IsPE
26 }
27
28 rule get_systemlang_process {
29     strings:
30         $lang = "GetSystemDefaultLangID"
31         $process = "CreateToolhelp32Snapshot"
32     condition:
33         ($lang or $process) and IsPE
34 }
35
36 rule sleep {
37     strings:
38         $sleep = "sleep" nocase
39     condition:
40         $sleep and IsPE
41 }
42
43 rule Internet {
44     strings:
45         $a = "IPPROTO_TCP"
46         $b = "SOCK_STREAM"
47         $c = "AF_INET"
48     condition:
49         ($a or $b or $c) and IsPE
50 }
51
52 rule VM {
```

```
53 strings:
54     $VM = "VMXh"
55 condition:
56     $VM and IsPE
57 }
```

使用规则进行，结果如下：

```
PS C:\Users\王峥\Desktop\恶意代码\上机实验样本> ./yara64 lab5.yar lab5
get lab5\Lab05-01.dll
remote_shell_session lab5\Lab05-01.dll
get_systemlang_process lab5\Lab05-01.dll
sleep lab5\Lab05-01.dll
```

图 3.39: yara 规则进行检测

最后两个 rule，即 Internet 和 VM 规则没有被扫描出来，可能是因为本身程序中不含有这些 rule 中的字符串。rule Internet 中的字符串是我们在问题 16 中的“Use Standard Symbolic Constant”替换出来的；rule VM 中的“VMXh”是我们在问题 17 中将十六进制串变为字符串得到的；上述 rule 中的字符串其实在原文中并没有体现，所以我们可以使用其原文中的地址/十六进制串/密文等进行判断，其余规则扫描成功。

3.3 IDA python

编写自动化 python 脚本，实现以下功能：

- 来到指定地址，并统计参数个数
- 找到指定函数，打印输出其汇编语句
- 搜索与遍历字符串
- 使用交叉引用查找函数或字符串调用关系

结合以上分析，编写规则如下：

Yara 规则

```
1 import idutils
2 sea = ScreenEA()
3
4 for i in range(0x00,0x50):
5     b = Byte(sea+i)
6     decoded_byte = b ^ 0x55
7     PatchByte(sea+i,decoded_byte)
8
9 print(idutils.Functions())
10
11 for i in idutils.Segments():
12     print(idc.SegName(i),idc.SegStart(i),idc.SegEnd(i))
```

```
13
14 for i in idutils.Functions():
15     flags = idc.GetFunctionFlags(i)
16     if flags & FUNC_LIB or flags & FUNC_THUNK:
17         continue
18     dism_addr = list(idutils.FuncItems(i))
19     for line in dism_addr:
20         j = idc.GetMnem(line)
21         if j == 'call':
22             op = idc.GetOpType(line, 0)
23             if op == o_reg:
24                 print("0x%x%s"%(line, idc.GetDisasm(line)))
```

运行结果如下:

```
-----
<generator object Functions at 0x031F8D78>
('.text', 268439552, 268525568)
('.idata', 268525568, 268526628)
('.rdata', 268526628, 268537856)
('.data', 268537856, 269037568)
('xdoors_d', 269037568, 269049856)
0x1000104ccall esi ; _stricmp
0x10001061ccall esi ; _stricmp
0x10001100ccall esi ; strchr
0x1000110fcall esi ; strchr
0x1000111ecall esi ; strchr
0x1000112dcall esi ; strchr
0x10001154ccall esi ; strchr
0x10001166ccall esi ; strchr
0x1000117bcall esi ; strchr
0x100011f1ccall esi ; strchr
0x10001203ccall esi ; strchr
0x1000121ccall esi ; strchr
0x10001289ccall esi ; strchr
0x1000129bccall esi ; strchr
0x100012b8ccall esi ; strchr
0x100013f1ccall esi ; strchr
0x10001400ccall esi ; strchr
0x1000140fccall esi ; strchr
0x1000141ecall esi ; strchr
0x10001445ccall esi ; strchr
0x10001457ccall esi ; strchr
0x1000146ccall esi ; strchr
0x100014e2ccall esi ; strchr
0x100014f4ccall esi ; strchr
0x1000150dcall esi ; strchr
0x1000157acall esi ; strchr
0x1000158ccall esi ; strchr
0x100015a9ccall esi ; strchr
0x100016f3ccall eax
0x1000179bccall esi ; strncpy
0x100017adccall esi ; strncpy
0x100017d6ccall esi ; strncpy
0x100017e8ccall esi ; strncpy
0x10001827ccall esi ; strncpy
0x10001839ccall esi ; strncpy
```

图 3.40: IDA Python 运行部分截图

上述的 IDA python 代码返回了 IDA 识别出的所有函数入口点列表; 遍历所有的指令, 所有的交叉引用地址, 还有搜索所有的代码和数据; 输出调用 call 指令的寄存器。

4 实验结论及心得体会

4.1 实验结论

本次实验我们主要是使用 IDA Pro 和 IDA Python 工具对恶意 DLL 文件进行逆向分析, 通过之前我们对 IDA 工具的了解与学习, 我们成功找到了 DllMain 函数的地址, 主要用于初始化和清理

DLL。再使用了 IDA Pro 的 Imports 窗口查找了导入函数的地址和名称，特别关注了 `gethostbyname` 函数，助于我们理解 DLL 调用的外部函数。同时通过追踪 `gethostbyname` 函数调用，找到调用位置和参数，了解恶意代码的功能。通过对全局变量和局部变量的分析，了解其作用。再查找和理解了对 Windows API 函数的直接调用，如 `Sleep` 和 `socket`，分析了它们的参数和功能。最后通过上述的一系列分析，编写 IDA Python 的脚本，实现自动化分析。

4.2 心得体会

本次实验不同之前在逆向分析课上单纯使用 IDA 进行分析，我们深入的了解了逆向工程的基本原理和工具，更细致的分析了程序的每个细节，包括函数调用、字符串、全局变量等，这有助于我们建立起对恶意代码的全面理解。最重要的是，我们使用 IDA Python 脚本实现自动化分析，从而加速分析速度，提升分析效率。