

南开大学

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：2113997

姓名：齐明杰

班级：信安2班

1 实验目的

通过使用IDA PRO分析恶意代码来熟悉对IDA的使用，以及对IDA Python脚本的使用。

2 实验原理

IDA Pro 是一款著名的交互式反汇编器和调试器。它被广大安全研究员、逆向工程师和黑客所使用，主要用于分析二进制程序，从而理解它的功能和行为。IDA Pro 支持多种处理器和操作系统，并为多种文件格式提供深入的分析。其中的交互式反汇编器可以将二进制代码转换成汇编代码，使其更易于人类理解。

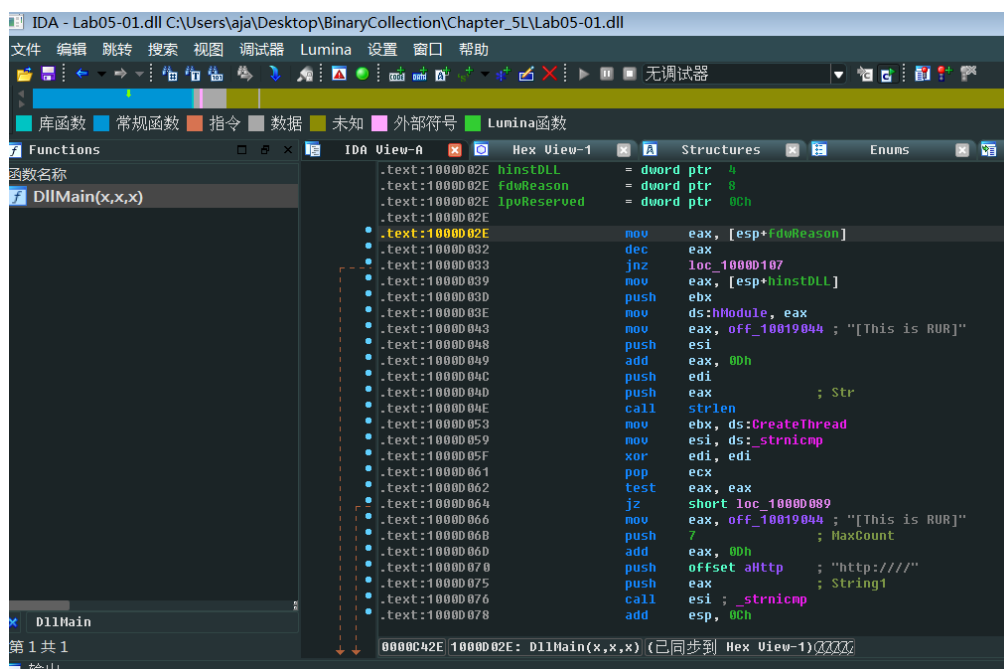
IDA Python 是一个在 IDA Pro 中集成的 Python 插件，允许用户使用 Python 脚本来扩展 IDA 的功能和进行自动化操作。这为用户提供了一个非常强大的工具，使他们能够编写自定义的脚本来自动进行复杂的分析任务，而不是手动执行这些任务。通过使用 IDA Python，用户可以更加高效地进行反汇编和二进制分析工作。

3 实验过程

3.1 Lab05-01.dll

- Q1: DLLMain 的地址是什么？

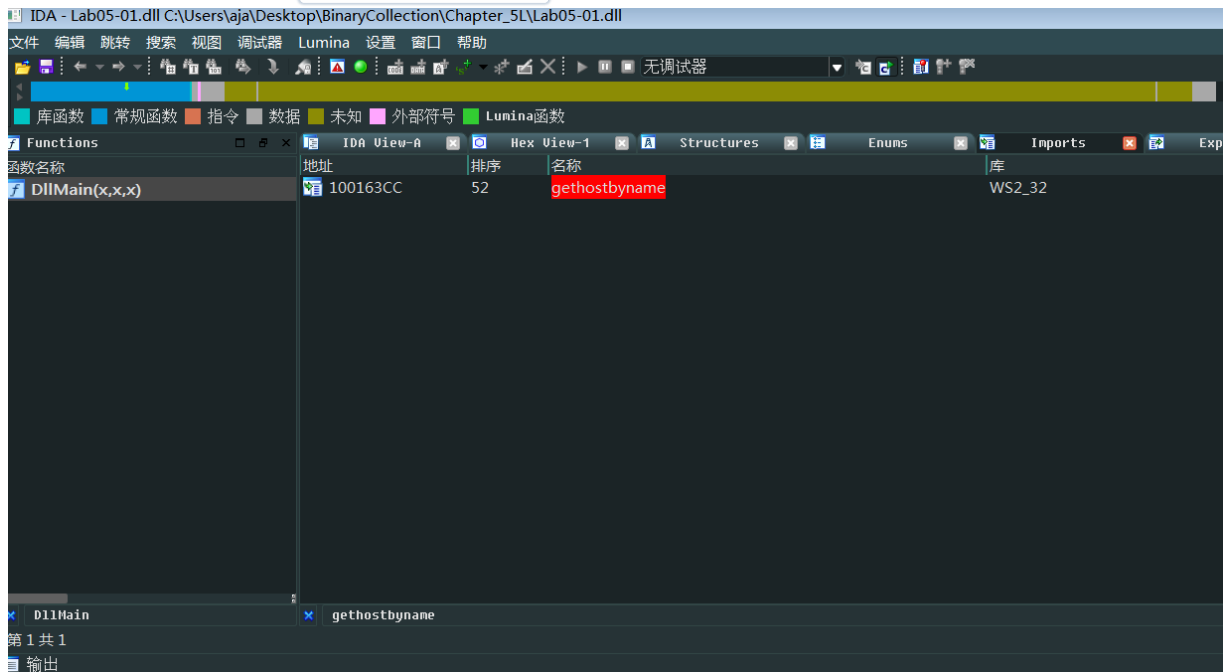
使用IDA PRO加载 **Lab05-01.dll**，在左侧函数搜索 **"DllMain"**，可以得到其地址：



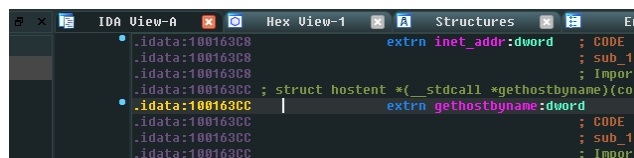
如图，地址为0x1000D02E，位于text节。

- Q2: 使用Imports窗并浏览到gethostbyname，导入函数定位到什么地址？

在Imports窗口内搜索 "gethostbyname"，可以找到此导入函数的地址：



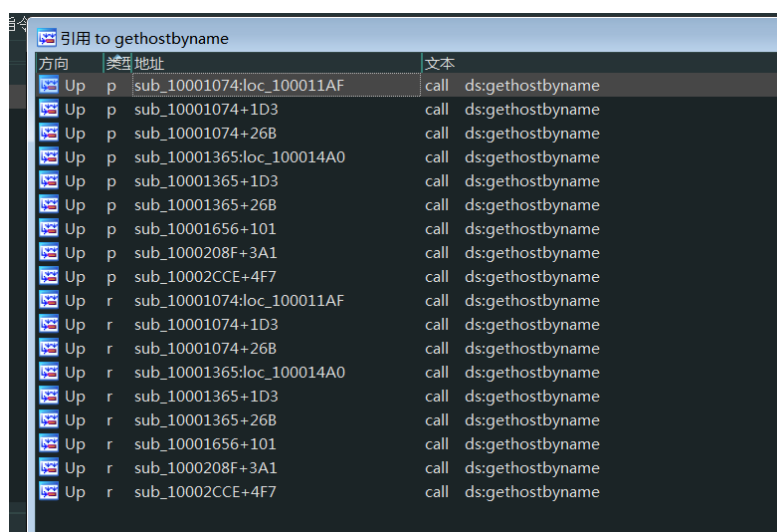
地址为0x100163CC。双击进入其所在地址：



可以发现这个地址位于idata节。

- Q3: 有多少函数调用了 gethostbyname?

使用快捷键Ctrl+x，查看gethostbyname的交叉引用：



可以发现一共18个结果，但仔细观察发现每个地址被查出来两次，类型分别为r和p，即读取和调用，因此有9个函数调用了gethostbyname。

- Q4: 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?

使用G快捷键跳转到0x10001757处, 如下图所示:

注意到其上方几个汇编语句:

```

1 | .text:1000174E      mov     eax, off_10019040 ; "[This is
   | RDO]pics.practicalmalwareanalys"...
2 | .text:10001753      add     eax, 0Dh
3 | .text:10001756      push    eax                ; name
4 | .text:10001757      call    ds:gethostbyname

```

其中地址0x10019040存放的是字符串 `[This is RDO]pics.practicalmalwareanalys.com` 的地址:

双击可以进入该字符串真正存放处(0x10019194):

在 `mov eax, off_10019040` 后, `eax` 的值为字符串的首地址, 并且发现 `[This is RDO]` 字符串的长度正好是13, 因此在 `add eax, 0Dh` 后, `eax` 指向字符串 `pics.practicalmalwareanalysis.com`, 故该域名的DNS请求将被触发。

- Q5: IDA Pro识别了在0x10001656处的子过程中的多少个局部变量?

跳转到该地址出, 可以看到许多变量:

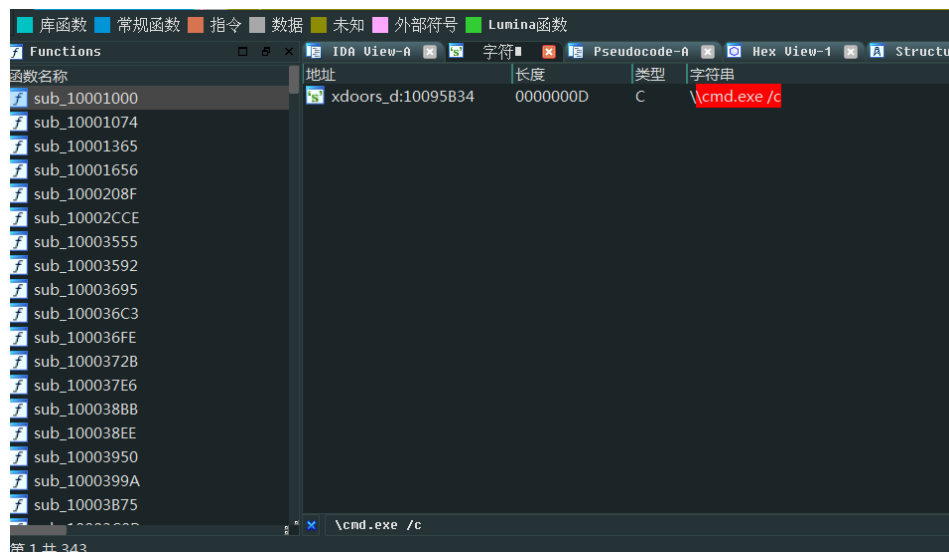
其中，偏移为负值的为局部变量，即除了最后一个，均为识别出的局部变量，共23个。

- Q6: IDA Pro识别了在0x10001656处的子过程中的多少个参数?

接上图，偏移为正值的为传递的参数，因此最后一个 `lpThreadParameter` 为识别出的参数：

- Q7: 使用Strings 窗口，来在反汇编中定位字符串\cmd.exe /c。它位于哪?

在Strings窗口中搜索字符串 `\cmd.exe /c`，可以得到结果：



它位于地址 `0x10095B34` 处，双击跟随，可以查看其及附近的字符串：

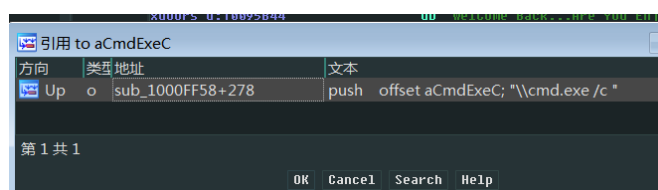
```

xdoors_d:10095810 align 10h
xdoors_d:10095820 ; char aCommandExeC[]
xdoors_d:10095820 aCommandExeC db '\command.exe /c ',0 ; DATA XREF: sub_1000FF58:loc_100101D7↑o
xdoors_d:10095831 align 4
xdoors_d:10095834 aCmdExeC db '\cmd.exe /c ',0 ; DATA XREF: sub_1000FF58+278↑o
xdoors_d:10095841 align 4
xdoors_d:10095844 ; char aHiMasterDDDDDD[]
xdoors_d:10095844 aHiMasterDDDDDD db 'Hi,Master [%d/%d/%d %d:%d:%d]',00h,00h ; DATA XREF: sub_1000FF58+145↑o
xdoors_d:10095844 db 'Welcome Back...Are You Enjoying Today?',00h,00h
xdoors_d:10095844 db 00h,00h
xdoors_d:10095844 db 'Machine UpTime [%-.2d Days %-.2d Hours %-.2d Minutes %-.2d Second'
xdoors_d:10095844 db 'ds]',00h,00h
xdoors_d:10095844 db 'Machine IdleTime [%-.2d Days %-.2d Hours %-.2d Minutes %-.2d Second'
xdoors_d:10095844 db 'nds]',00h,00h
xdoors_d:10095844 db 00h,00h
xdoors_d:10095844 db 'Encrypt Magic Number For This Remote Shell Session [0x%02x]',00h,0
xdoors_d:10095844 db 00h,00h,0
xdoors_d:10095C5C ; char asc_10095C5C[]
xdoors_d:10095C5C asc_10095C5C db '>',0 ; DATA XREF: sub_1000FF58+4B↑o
xdoors_d:10095C5C ; sub_1000FF58+3E1↑o
xdoors_d:10095C5E align 400h
xdoors_d:10095C5E ends

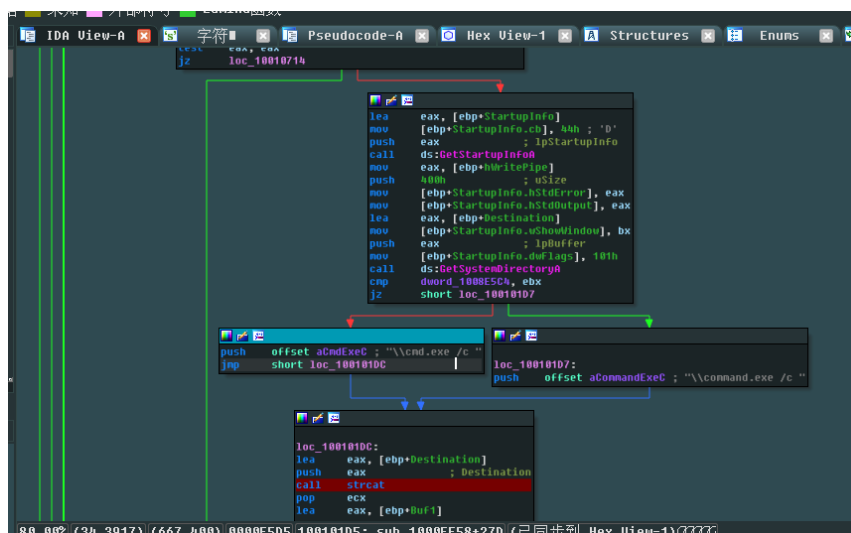
```

- Q8: 在引用\cmd.exe/c的代码所在的区域发生了什么？

查看该字符串的交叉引用列表：



进入该指令所在地址：



观察其控制流图中上下文，发现字符串 `Hi,Master [%d/%d/%d %d:%d:%d]` 等的使用，进入字符串所在之处查看：

```

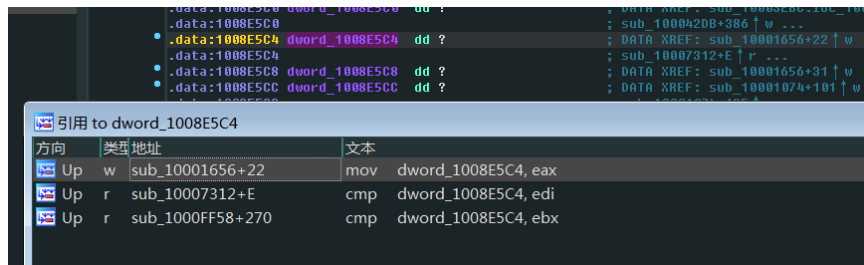
xdoors_d:10095844 ; char aHiMasterDDDDDD[]
xdoors_d:10095844 aHiMasterDDDDDD db 'Hi,Master [%d/%d/%d %d:%d:%d]',00h,00h ; DATA XREF: sub_1000FF58+145↑o
xdoors_d:10095844 db 'Welcome Back...Are You Enjoying Today?',00h,00h
xdoors_d:10095844 db 00h,00h
xdoors_d:10095844 db 'Machine UpTime [%-.2d Days %-.2d Hours %-.2d Minutes %-.2d Second'
xdoors_d:10095844 db 'ds]',00h,00h
xdoors_d:10095844 db 'Machine IdleTime [%-.2d Days %-.2d Hours %-.2d Minutes %-.2d Second'
xdoors_d:10095844 db 'nds]',00h,00h
xdoors_d:10095844 db 00h,00h
xdoors_d:10095844 db 'Encrypt Magic Number For This Remote Shell Session [0x%02x]',00h,00h
xdoors_d:10095844 db 00h,00h,0

```

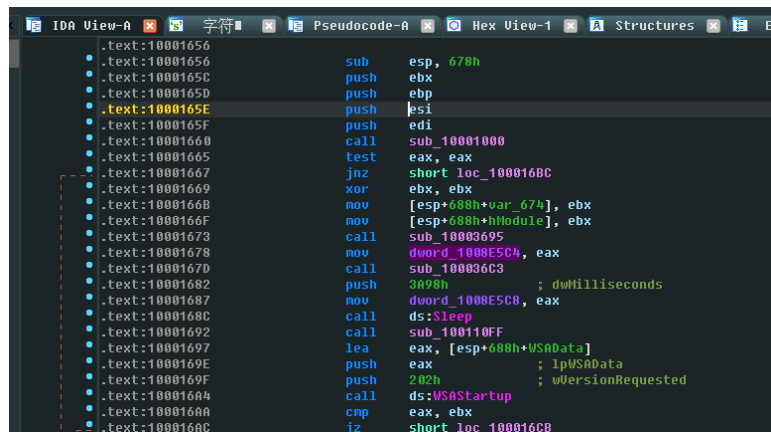
发现了 `Remote Shell Session` 等字样，推测这一片代码区域开启了一段远程Shell会话。

- Q9: 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。)

查看 `dword_1008E5C4` 的交叉引用列表:



发现有两个读操作, 一个写操作, 双击进入写操作所在代码:

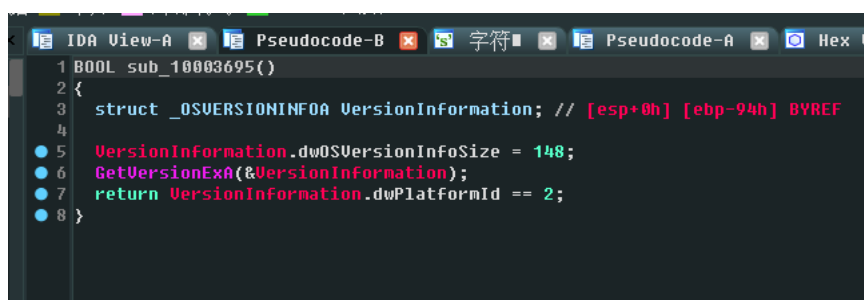


可以发现, 其值就是函数 `sub_10003695` 在此处的返回值。

进入函数查看:



按F5进行反编译, 得到C语言代码:

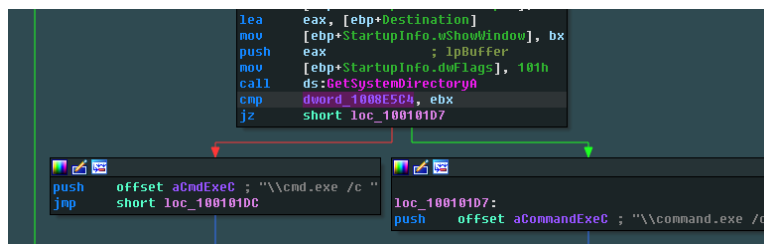


`GetVersionExA` 函数用于获取当前操作系统的信息，因此这个函数的作用是获取该电脑操作系统的版本，然后对比 `VersionInformation.dwPlatformId` 是否为2(即 `VER_PLATFORM_WIN32_NT`)。

因此可以总结恶意代码设置 `dword_1008E5C4` 的流程：

通过函数 `sub_10003695` 获取该电脑是否是 win32 NT 系统，若是，则设置 `dword_1008E5C4` 为 1，否则设置为 0。

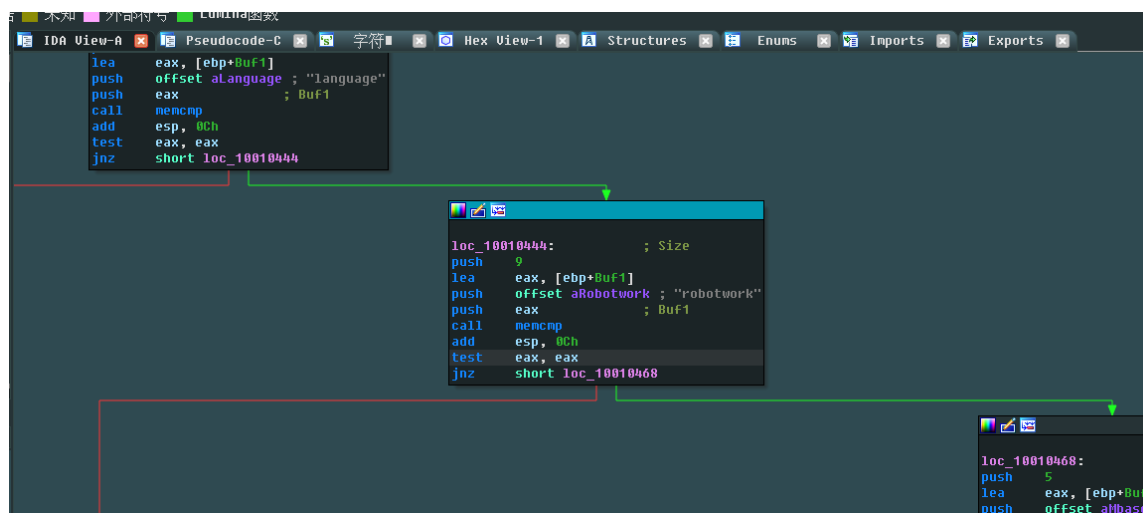
通过再次查看 `dword_1008E5C4` 的交叉引用，进入读取该全局变量的代码：



综合上述分析，不难看出代码判断操作系统版本是为了选择使用 `cmd.exe` 还是 `command.exe`。

- Q10: 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？

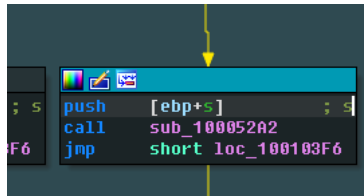
搜索字符串 `robotwork`，找到相关指令：



可以看到若字符串比较成功，则 `memcmp` 返回 0，即 `eax` 为 0，此时

```
1 | test eax, eax
```

这个指令不修改操作数，只修改标志寄存器，因此零标志位置 1，即 `ZF=1`，因此 `jnz` 走红色线，查看下方代码：



进入函数 `sub_100052A2` 并反编译查看：

```

9 char v8; // [esp+403h] [ebp-200h]
10 BYTE Data[512]; // [esp+404h] [ebp-20Ch] BYREF
11 DWORD cbData; // [esp+604h] [ebp-Ch] BYREF
12 DWORD Type; // [esp+608h] [ebp-8h] BYREF
13 HKEY phkResult; // [esp+60Ch] [ebp-4h] BYREF
14
15 memset(Buffer, 0, sizeof(Buffer));
16 memset(Data, 0, sizeof(Data));
17 v7 = 0;
18 v8 = 0;
19 if ( RegOpenKeyEx(HKEY_LOCAL_MACHINE, aSoftwareMicros, 0, 0xF003Fu, &phkResult) )
20 return RegCloseKey(phkResult);
21 if ( !RegQueryValueEx(phkResult, aWorktime, 0, &Type, Data, &cbData) )
22 {
23     v2 = atoi((const char *)Data);
24     sprintf(Buffer, "\r\n\r\n[Robot_WorkTime :] %d\r\n\r\n", v2);
25     v3 = strlen(Buffer);
26     sub_100038EE(s, (int)Buffer, v3);
27 }
28 memset(Data, 0, sizeof(Data));
29 if ( !RegQueryValueEx(phkResult, aWorktimes, 0, &Type, Data, &cbData) )
30 {
31     v4 = atoi((const char *)Data);
32     sprintf(Buffer, "\r\n\r\n[Robot_WorkTimes:] %d\r\n\r\n", v4);
33     v5 = strlen(Buffer);
34     sub_100038EE(s, (int)Buffer, v5);
35 }
36 return RegCloseKey(phkResult);
37 }

```

发现其使用了 `RegOpenKeyEx`，`RegCloseKey`，`RegQueryValueEx` 等函数，用于修改注册表的值。另外发现其调用 `sub_100038EE` 函数，也查看其反编译代码：

```

5 int v5; // edx
6 char *v6; // esi
7 char *v7; // ecx
8 int v8; // eax
9 int v9; // eax
10 int v10; // edi
11
12 v3 = len;
13 v4 = (char *)malloc(len + 1);
14 v5 = 0;
15 v6 = v4;
16 if ( len > 0 )
17 {
18     v7 = v4;
19     v8 = a2 - (_DWORD)v4;
20     v5 = len;
21     do
22     {
23         *v7 = dword_1008E5D0 + v7[v8];
24         ++v7;
25         --len;
26     } while ( len );
27 }
28 v6[v5] = 0;
29 v9 = send(s, v6, v3, 0);
30 v10 = -1;
31 if ( v9 != -1 )
32 v10 = v9;
33 free(v6);
34 return v10;
35 }
36 }

```

发现其调用了 `send` 和 `free` 函数，用于发送 `"\r\n\r\n[Robot_WorkTime :] %d\r\n\r\n"` 等信息，因此总结其功效是修改注册表的值，并发送注册表键 `WorkTimes` 对应的值。

• Q11: PSLIST导出函数做了什么？

我们可以在导出函数列表找到PSLIST：

名称	地址	排序
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	[main entry]

双击进入其反汇编代码：

```

1 int __stdcall PSLIST(int a1, int a2, char *Str, int a4)
2 {
3     int result; // eax
4
5     dword_1008E58C = 1;
6     result = sub_100036C3();
7     if ( result )
8     {
9         if ( strlen(Str) )
10            result = sub_1000664C(0, Str);
11        else
12            result = sub_10006518(0);
13    }
14    dword_1008E58C = 0;
15    return result;
16 }

```

发现其调用了若干个函数：sub_100036C3, sub_1000664C, sub_10006518，依次进入查看反汇编：

1. sub_100036C3

```

1 BOOL sub_100036C3()
2 {
3     struct _OSVERSIONINFOA VersionInformation; // [esp+0h] [ebp-94h] BYREF
4
5     VersionInformation.dwOSVersionInfoSize = 148;
6     GetVersionExA(&VersionInformation);
7     return VersionInformation.dwPlatformId == 2 && VersionInformation.dwMajorVersion >= 5;
8 }

```

不难看出这个函数的作用是判断操作系统是否为Win32系统，以及版本是否不小于Win2000，如果是则返回1。

2. sub_1000664C

```

1 int __usercall sub_1000664C@eax(char a1@sil, SOCKET s, char *Str)
2 {
3     DWORD LastError; // eax
4     size_t v5; // eax
5     HANDLE v6; // ebx
6     DWORD v7; // eax
7     int v8[1024]; // [esp+8h] [ebp-1634h] BYREF
8     char Buffer[1024]; // [esp+1008h] [ebp-634h] BYREF
9     __int6 v10; // [esp+1405h] [ebp-237h]
10    char v11; // [esp+1407h] [ebp-235h]
11    char v12[260]; // [esp+1408h] [ebp-234h] __int6 v10; // [esp+1405h] [ebp-237h]
12    PROCESSENTRY32 pe; // [esp+150Ch] [ebp-130h] BYREF
13    char v14[4]; // [esp+1634h] [ebp-8h] BYREF
14    HANDLE hSnapshot; // [esp+1638h] [ebp-4h]
15
16    memset(Buffer, 0, sizeof(Buffer));
17    v10 = 0;
18    v11 = 0;
19    memset(&pe, 0, sizeof(pe));
20    memset(v8, 0, sizeof(v8));
21    hSnapshot = CreateToolhelp32Snapshot(2u, 0);
22    if ( hSnapshot == (HANDLE)-1 )
23    {
24        LastError = GetLastError();
25        sprintf(Buffer, "CreateToolhelp32Snapshot Fail:Error%d", LastError);
26        sub_100038B8(s, Buffer);
27        return 1;
28    }
29    else

```

```

37 {
38     do
39     {
40         v5 = strlen(Str);
41         if ( !strnicmp(pe.szExeFile, Str, v5) )
42         {
43             v6 = OpenProcess(0x410u, 0, pe.th32ProcessID);
44             EnumProcessModules(v6, v8, 4096, v14);
45             memset(v12, 0, sizeof(v12));
46             GetModuleFileNameEx(v6, v8[0], v12, 260);
47             sprintf(Buffer, "\\A\\%16d%-20s%d", pe.th32ProcessID, pe.szExeFile, pe.cntThreads);
48             sub_10003880(s, Buffer);
49             sprintf(Buffer, "\\A\\%16s]", v12);
50             sub_10003880(s, Buffer);
51             if ( dword_1000E58C )
52                 sub_1000628C(a16d28ds, pe.th32ProcessID);
53             CloseHandle(v6);
54         }
55     } while ( Process32Next(hSnapshot, &pe) );
56 }
57 else
58 {
59     v7 = GetLastError();
60     sprintf(Buffer, "\\A\\Process32First() Fail:Error %d", v7);
61     sub_10003880(s, Buffer);
62 }
63 sub_10003880(s, :, Buffer);
64 CloseHandle(hSnapshot);
65

```

3. sub_10006518

```

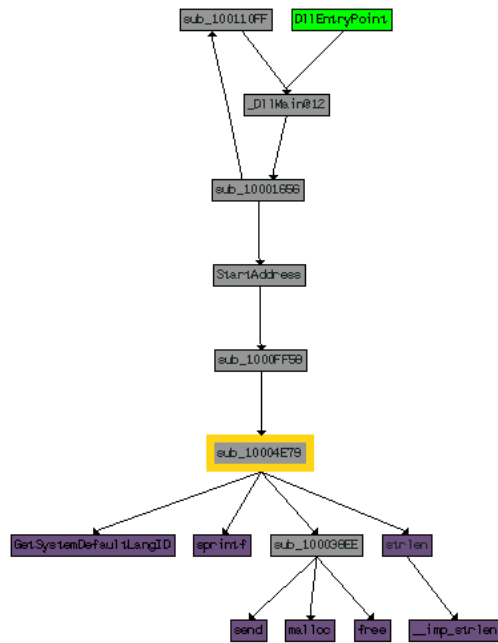
6 int v4[1024]; // [esp+0h] [ebp-1530h] BYREF
7 char v5[1024]; // [esp+100h] [ebp-530h] BYREF
8 int v6; // [esp+140h] [ebp-130h]
9 char v7; // [esp+140h] [ebp-130h]
10 PROCESSENTRY32 pe; // [esp+140h] [ebp-130h] BYREF
11 char v9[4]; // [esp+1530h] [ebp-8h] BYREF
12 HANDLE hSnapshot; // [esp+1530h] [ebp-4h]
13
14 memset(v4, 0, sizeof(v4));
15 memset(v5, 0, sizeof(v5));
16 v6 = 0;
17 v7 = 0;
18 hSnapshot = CreateToolhelp32Snapshot(2u, 0);
19 if ( hSnapshot != (HANDLE)-1 )
20 {
21     pe.dwSize = 296;
22     if ( dword_1000E58C )
23         sub_1000628C(aProcessidProce, v9);
24     for ( i = Process32First(hSnapshot, &pe); i; i = Process32Next(hSnapshot, &pe) )
25     {
26         v1 = OpenProcess(0x410u, 0, pe.th32ProcessID);
27         EnumProcessModules(v1, v4, 4096, v9);
28         memset(v5, 0, 0x100u);
29         GetModuleFileNameEx(v1, v4[0], v5, 1024);
30         if ( dword_1000E58C )
31             sub_1000628C(a16d28ds, pe.th32ProcessID);
32         CloseHandle(v1);
33     }
34 }
35 CloseHandle(hSnapshot);
36 return 0;
37 }

```

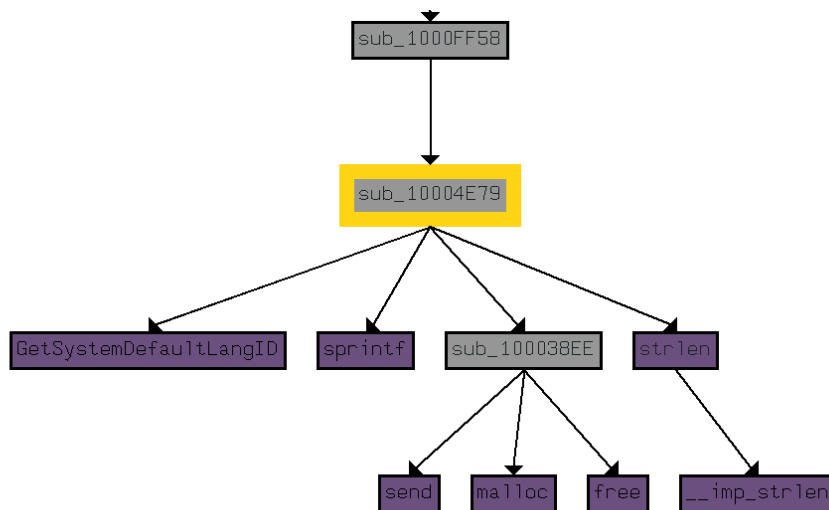
sub_1000664C 和 sub_10006518 两个函数都调用了 CreateToolhelp32Snapshot(), 并且都调用了与进程有关函数, 可以分析出其作用是获取进程列表, 然后通过socket发送各个进程的信息。

- Q12: 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?

跳转到0x10004E79, 通过 View -> Graphs -> User xrefs chart 得到交叉引用图如下:



其核心部分：

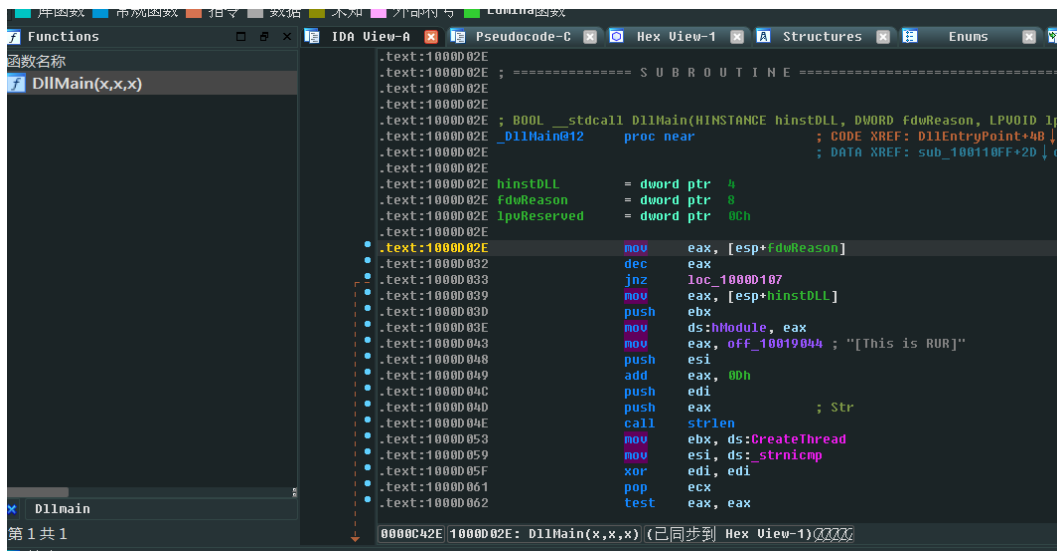


可以看到sub_10004E79调用了 `GetSystemDefaultLangID`, 这个函数用于获取系统的默认语言ID, 然后使用了socket函数 `send` 进行发送。

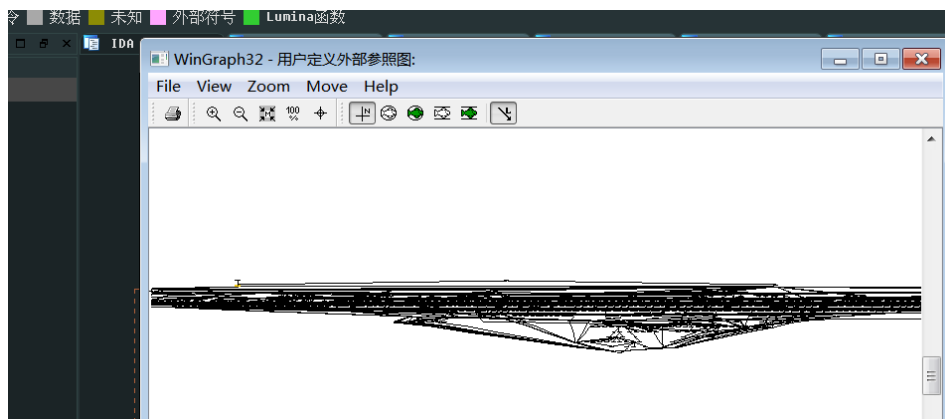
因此我们可以使用快捷键N, 重命名该函数为 `SendLanguageID`:

- Q13: DllMain 直接调用了多少个 Windows API? 多少个在深度为2时被调用?

转到DllMain的起始地址(0x1000D02E):

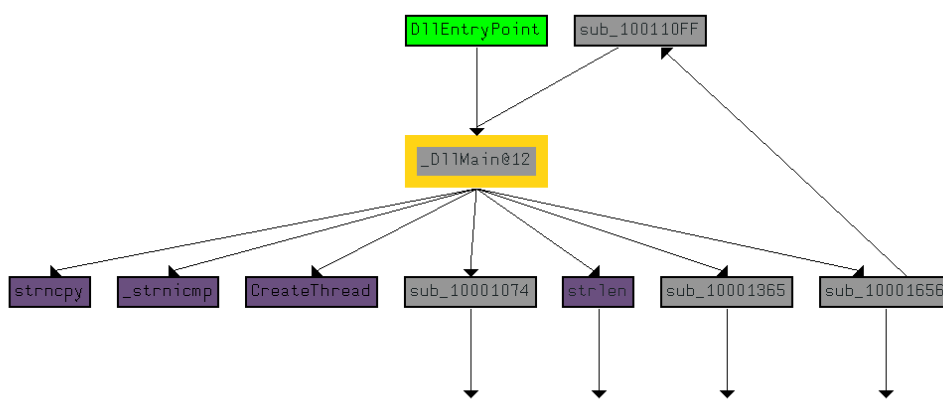


同样打开交叉引用图：



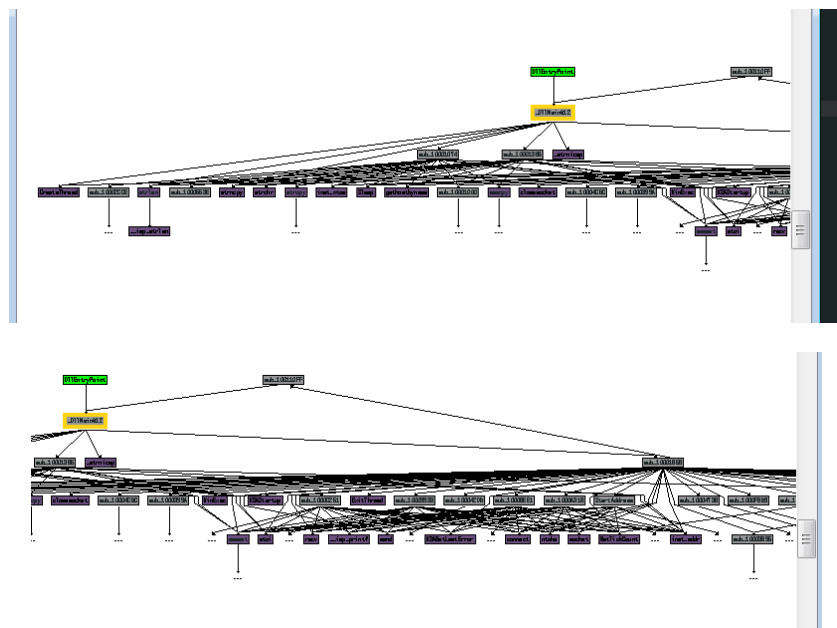
由于函数过多，我们需要设置一下生成交叉引用图的参数：

把递归深度设置为1：



这样就可以看到DllMain直接引用的API函数，为 `strcpy`，`_strnicmp`，`CreateThread`，`strlen`，共四个。

把递归深度设置为2：



如图，可以看到深度为2的API个数非常庞大。

- Q14: 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？

该地址附近的关键代码：

```

1 | .text:10001341      mov     eax, off_10019020 ; "[This is
   | CTI]30"
2 | .text:10001346      add     eax, 0Dh
3 | .text:10001349      push    eax                ; String
4 | .text:1000134A      call    ds:atoi
5 | .text:10001350      imul    eax, 3E8h
6 | .text:10001356      pop     ecx
7 | .text:10001357      push    eax                ; dwMilliseconds
8 | .text:10001358      call    ds:Sleep

```

在前两句后，eax指向字符串 "30"，然后调用 atoi 函数，即字符串转整数，将eax赋为30。

再将eax乘以0x3E8（为十进制的1000），故传入Sleep的参数为 $30 * 1000 = 3 * 10^4 \text{ ms} = 30 \text{ s}$ ，

因此将休眠30秒。

- Q15: 在0x10001701处是一个对 socket 的调用。它的3个参数是什么？

在该地址附近代码：

```

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

```

根据注释可以得到三个参数按顺序依次为 `af = 2`，`type = 1`，`protocol = 6`。

- Q16: 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?

分别对应MSDN上socket的符号常量, 修改如下图所示:

```
.text:100016FB ; sub_10001656+A09
.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      ;
```

- Q17: 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 指函数的交叉引用, 能发现进一步检测VMware的证据吗?

进入Search→Sequence of Bytes, 搜索0xED:



翻阅结果, 找到一个in指令:

地址	函数	指令
.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+var_1294], ebx
.text:10006318	sub_100062E9	call ???@YAPAXI@Z; operator new(uint)
.text:10006476	sub_100062E9	lea ecx, [ebp+var_1294]
.text:100064A9	sub_100062E9	push [ebp+var_1294]
.text:1000671B	sub_1000664C	call sub_1000620C
.text:10006C43	sub_10006BD5	jnz short loc_10006C31
.text:10006D15	sub_10006CA7	jnz short loc_10006D03
.text:10008667	sub_1000834E	call sub_100073DA

进入该指令附近:

```
.text:100061C8 ; __try { __except at loc_100061E1
.text:100061C8      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:100061C8      mov     ebx, 0
.text:100061D1      mov     ecx, 0Ah
.text:100061D6      mov     edx, 5658h
.text:100061D8      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:100061F6
```

使用快捷键R将564D5868h直接转化为字符串:

```

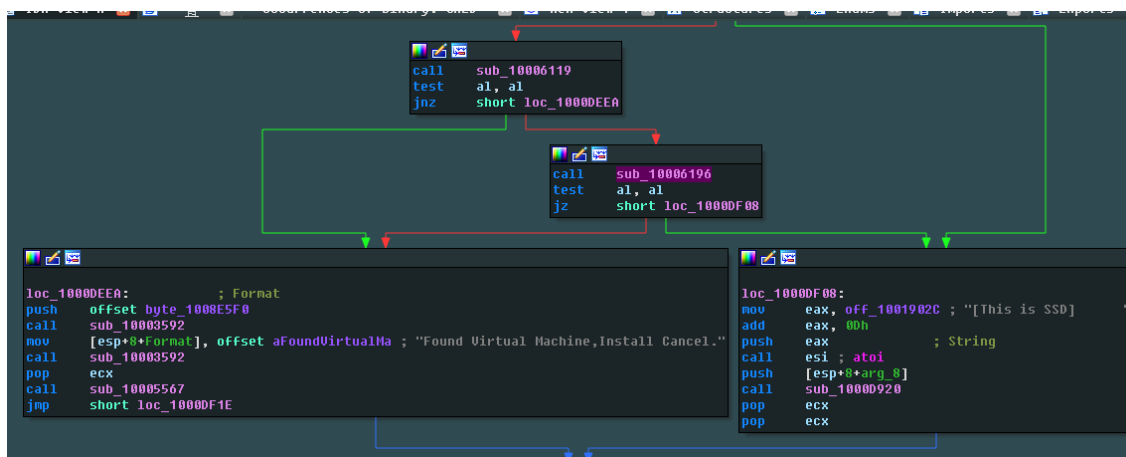
ext:100061C4      push     edx
ext:100061C5      push     ecx
ext:100061C6      push     ebx
ext:100061C7      mov     eax, 'VMXh'
ext:100061CC      mov     ebx, 0
ext:100061D1      mov     ecx, 0Ah
ext:100061D6      mov     edx, 5658h
ext:100061DB      in      eax, dx
ext:100061DC      cmp     ebx, 'VMXh'
ext:100061E2      setz    [ebp+var_1C]
ext:100061E6      pop     ebx
ext:100061E7      pop     ecx
ext:100061E8      pop     edx
ext:100061E9      jmp     short loc_100061F6
ext:100061EB ;

```

发现这个字符串就是VMXh，转到函数头，查看这个函数的交叉引用：

引用到 sub_10006196			
方向	类型	地址	文本
Do...	p	InstallRT+20	call sub_10006196
Do...	p	InstallSA+20	call sub_10006196
Do...	p	InstallSB+20	call sub_10006196

进入第一个：



发现其下面存在 "Found Virtual Machine, Install Cancel." 字符串，这直接证明了恶意代码中存在虚拟机检测。

- Q18: 将你的光标跳转到 0x1001D988 处，你发现了什么？

定位到题述位置，发现是一堆意义不明的字节序列：

```

.data:1001D988 db 20h ; -
.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 30h ; =
.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 ; &

```

解密该字节序列见Q19。

- Q19: 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?

运行Lab05-01.py, 结果报错了:

```
C:\Users\aja\Desktop\BinaryCollection\Chapter_5L\Lab05-01.py: name 'ScreenEA' is not defined
Traceback (most recent call last):
  File "C:\deSec
\IDA7.6\64_idapronw_hexarm64w_hexarmw_hexmipsw_hexppc64w_hexppcw_hexx64w_hexx8
\3\ida_idaapi.py", line 616, in IDAPython_ExecScript
    exec(code, g)
  File "C:/Users/aja/Desktop/BinaryCollection/Chapter_5L/Lab05-01.py", line 1, in <module>
    sea = ScreenEA()
NameError: name 'ScreenEA' is not defined
```

推测是使用的代码与我使用的IDAPython版本不符(IDAPython v7.4.0), 据此修改代码(不改变原意)如下:

```
1 import ida_bytes
2
3 ea = idc.get_screen_ea()
4
5 for i in range(0x00, 0x50):
6     b = ida_bytes.get_byte(ea + i) # 读取字节
7     decoded_byte = b ^ 0x55 # 解码字节
8     ida_bytes.patch_byte(ea + i, decoded_byte) # 打补丁
9
```

再次运行脚本, 成功修改字节数据,

Address	Hex	ASCII
1001D986	00	
1001D987	00	
1001D988	78h	x
1001D989	64h	d
1001D98A	6Fh	o
1001D98B	6Fh	o
1001D98C	72h	r
1001D98D	20h	
1001D98E	69h	i
1001D98F	73h	s
1001D990	20h	
1001D991	74h	t
1001D992	68h	h
1001D993	69h	i
1001D994	73h	s
1001D995	20h	
1001D996	62h	b
1001D997	61h	a
1001D998	63h	c
1001D999	68h	k
1001D99A	64h	d
1001D99B	6Fh	o
1001D99C	6Fh	o

- Q20: 将光标放在同一位置, 你如何将这个数据转成一个单一的ASCII字符串?

按A快捷键将其整合为字符串:

Address	Hex	ASCII
1001D984	00	
1001D985	00	
1001D986	00	
1001D987	00	
1001D988	xdoor is this backdoor, string decoded for Practical Malware Anal	
1001D989	00	
1001D98A	00	
1001D98B	00	
1001D98C	00	
1001D98D	00	
1001D98E	00	
1001D98F	00	
1001D990	00	
1001D991	00	
1001D992	00	
1001D993	00	
1001D994	00	
1001D995	00	
1001D996	00	
1001D997	00	
1001D998	00	
1001D999	00	
1001D99A	00	
1001D99B	00	
1001D99C	00	

为"xdoor is this backdoor, string decoded for practical Malware Analysis Lab :~1234"。

- Q21: 使用一个文本编辑器打开这个脚本。它是如何工作的?

```
1 | ea = idc.get_screen_ea()
```

这句代码获取当前光标所在地址。

```
1 | for i in range(0x00, 0x50):
2 |     b = ida_bytes.get_byte(ea + i) # 读取字节
3 |     decoded_byte = b ^ 0x55 # 解码字节
4 |     ida_bytes.patch_byte(ea + i, decoded_byte) # 打补丁
```

在连续50个字节内，读取每个字节，然后和0x55做异或操作，最后写入，完成补丁操作。

3.2 yara规则编写

综合以上，可以完成该恶意代码的yara规则编写：

```
1 | //首先判断是否为PE文件
2 | private rule IsPE
3 | {
4 |     condition:
5 |         filesize < 10MB and //小于10MB
6 |         uint16(0) == 0x5A4D and //"MZ"头
7 |         uint32(uint32(0x3C)) == 0x00004550 // "PE"头
8 | }
9 |
10 | //Lab05-01
11 | rule lab5_1
12 | {
13 |     strings:
14 |         $s1 = "pics.praticalmalwareanalysis.com"
15 |         $s2 = "cmd.exe /c"
16 |         $s3 = "Remote Shell Session"
17 |         $s4 = "robotwork"
18 |     condition:
19 |         IsPE and $s1 and $s2 and $s3 and $s4
20 | }
```

3.3 IDA Python脚本编写

我们可以编写如下Python脚本来辅助分析：

```
1 | import idutils
2 | import idc
3 | import idaapi
```

```

4
5 def is_jump_or_call_with_register(ea):
6     """
7     检查给定地址的助记符是否为 'jmp' 或 'call' 且操作数为寄存器类型
8     """
9     mnemonic = idc.print_insn_mnem(ea)
10    if mnemonic not in ['jmp', 'call']:
11        return False
12    opnd_type = idc.get_operand_type(ea, 0)
13    # 确保操作数是寄存器类型
14    return opnd_type in [idaapi.o_reg, idaapi.o_phrase, idaapi.o_displ]
15
16 def is_library_function(func_ea):
17     """
18     检查给定地址的函数是否为库函数
19     """
20    flags = idc.get_func_attr(func_ea, idc.FUNCATTR_FLAGS)
21    return flags & idaapi.FUNC_LIB
22
23 def main():
24    for func in idutils.Functions():
25        # 排除库函数
26        if is_library_function(func):
27            continue
28
29        # 遍历函数内的所有指令
30        ea = func
31        while ea != idaapi.BADADDR and ea < idc.find_func_end(func):
32            # 如果是跳转或调用并且操作数是寄存器类型
33            if is_jump_or_call_with_register(ea):
34                print("Address: 0x{:X}, Instruction: {}".format(ea,
35idc.generate_disasm_line(ea, 0)))
36
37            # 移动到下一个指令
38            ea = idc.next_head(ea)
39
40 if __name__ == '__main__':
41     main()

```

脚本流程：

1. **遍历所有函数：**脚本首先获取当前二进制文件中的所有函数。
2. **排除库函数：**对于每一个找到的函数，脚本检查是否为一个库函数。库函数通常是预编译的，与特定的应用程序逻辑无关，所以我们选择忽略它们。

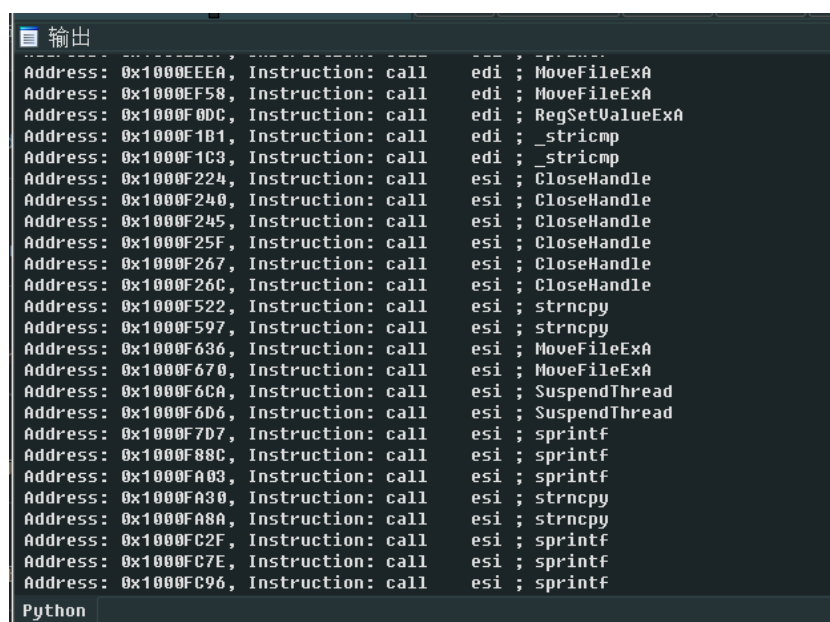
3. 遍历函数内的所有指令：对于每个非库函数，脚本将遍历该函数中的每一条指令。

4. 查找特定的指令：脚本查找具有以下特征的指令：

- 助记符为 `jmp` 或 `call`。
- 该指令的操作数是寄存器类型。这意味着指令是跳转或调用一个寄存器中的地址，而不是一个固定的地址或内存位置。

5. 输出匹配的指令：对于每个匹配的指令，脚本将输出该指令的地址和反汇编。

运行该IDA Python脚本，可以得到使用了call或jmp的指令，并且其操作数为寄存器类型：



```
输出
-----
Address: 0x1000EEEA, Instruction: call    edi ; MoveFileExA
Address: 0x1000EF58, Instruction: call    edi ; MoveFileExA
Address: 0x1000F0DC, Instruction: call    edi ; RegSetValueExA
Address: 0x1000F1B1, Instruction: call    edi ; _stricmp
Address: 0x1000F1C3, Instruction: call    edi ; _stricmp
Address: 0x1000F224, Instruction: call    esi ; CloseHandle
Address: 0x1000F240, Instruction: call    esi ; CloseHandle
Address: 0x1000F245, Instruction: call    esi ; CloseHandle
Address: 0x1000F25F, Instruction: call    esi ; CloseHandle
Address: 0x1000F267, Instruction: call    esi ; CloseHandle
Address: 0x1000F26C, Instruction: call    esi ; CloseHandle
Address: 0x1000F522, Instruction: call    esi ; strncpy
Address: 0x1000F597, Instruction: call    esi ; strncpy
Address: 0x1000F636, Instruction: call    esi ; MoveFileExA
Address: 0x1000F670, Instruction: call    esi ; MoveFileExA
Address: 0x1000F6CA, Instruction: call    esi ; SuspendThread
Address: 0x1000F6D6, Instruction: call    esi ; SuspendThread
Address: 0x1000F7D7, Instruction: call    esi ; sprintf
Address: 0x1000F88C, Instruction: call    esi ; sprintf
Address: 0x1000FA03, Instruction: call    esi ; sprintf
Address: 0x1000FA30, Instruction: call    esi ; strncpy
Address: 0x1000FA8A, Instruction: call    esi ; strncpy
Address: 0x1000FC2F, Instruction: call    esi ; sprintf
Address: 0x1000FC7E, Instruction: call    esi ; sprintf
Address: 0x1000FC96, Instruction: call    esi ; sprintf
Python
```

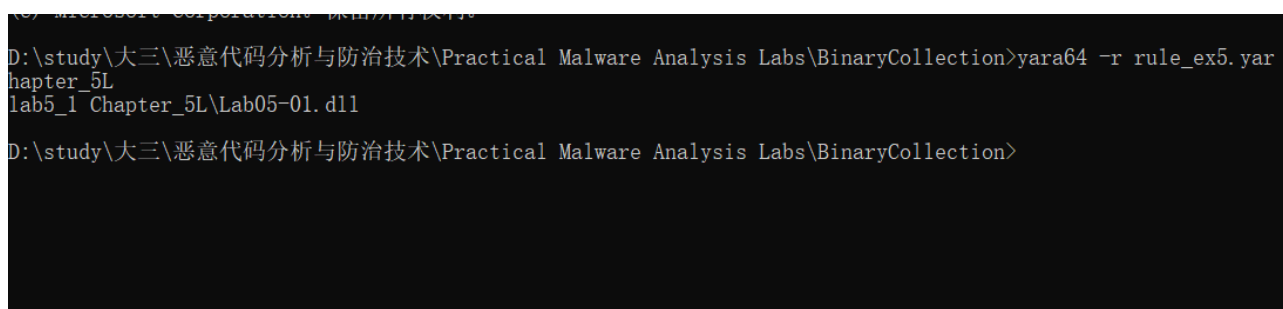
可以看到许多指令被成功筛选了出来。

4 实验结论及心得体会

把上述Yara规则保存为 `rule_ex5.yar`，然后在Chapter_5L上一个目录输入以下命令：

```
1 | yara64 -r rule_ex5.yar Chapter_5L
```

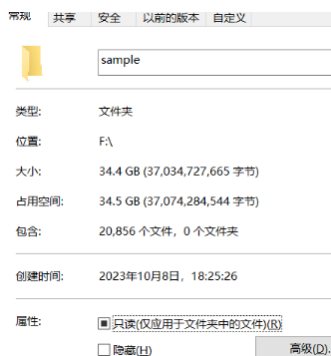
结果如下，样本检测成功：



```
(C) Microsoft Corporation. 保留所有权利。
D:\study\大三\恶意代码分析与防治技术\Practical Malware Analysis Labs\BinaryCollection>yara64 -r rule_ex5.yar Chapter_5L
lab5_1 Chapter_5L\Lab05-01.d11
D:\study\大三\恶意代码分析与防治技术\Practical Malware Analysis Labs\BinaryCollection>
```

接下来对运行 `Scan.py` 获得的sample进行扫描。

sample文件夹大小为34.4GB，含有20856个可执行文件：



我们编写一个yara扫描脚本 `yara_unittest.py` 来完成扫描：

```
1 import os
2 import yara
3 import datetime
4
5 # 定义YARA规则文件路径
6 rule_file = './rule_ex5.yar'
7
8 # 定义要扫描的文件夹路径
9 folder_path = './sample/'
10
11 # 加载YARA规则
12 try:
13     rules = yara.compile(rule_file)
14 except yara.SyntaxError as e:
15     print(f"YARA规则语法错误: {e}")
16     exit(1)
17
18 # 获取当前时间
19 start_time = datetime.datetime.now()
20
21 # 扫描文件夹内的所有文件
22 scan_results = []
23
24 for root, dirs, files in os.walk(folder_path):
25     for file in files:
26         file_path = os.path.join(root, file)
27         try:
28             matches = rules.match(file_path)
29             if matches:
30                 scan_results.append({'file_path': file_path, 'matches':
31 [str(match) for match in matches]})
32         except Exception as e:
33             print(f"扫描文件时出现错误: {file_path} - {str(e)}")
34
35 # 计算扫描时间
```

```

35 end_time = datetime.datetime.now()
36 scan_time = (end_time - start_time).seconds
37
38 # 将扫描结果写入文件
39 output_file = './scan_results.txt'
40
41 with open(output_file, 'w') as f:
42     f.write(f"扫描开始时间: {start_time.strftime('%Y-%m-%d %H:%M:%S')}\n")
43     f.write(f"扫描耗时: {scan_time}s\n")
44     f.write("扫描结果:\n")
45     for result in scan_results:
46         f.write(f"文件路径: {result['file_path']}\n")
47         f.write(f"匹配规则: {' , '.join(result['matches'])}\n")
48         f.write('\n')
49
50 print(f"扫描完成, 耗时{scan_time}秒, 结果已保存到 {output_file}")

```

运行得到扫描结果文件如下:

```

1 扫描开始时间: 2023-10-13 12:54:58
2 扫描耗时: 89s
3 扫描结果:
4 文件路径: ./sample/Lab05-01.dll
5 匹配规则: lab5_1

```

只有该恶意代码样本可执行文件被扫描识别, 共耗时89s。

心得体会: 通过此次实验, 我学会了IDA PRO的使用。这是一款逆向神器, 不仅能够查看反汇编代码, 还能进行反编译, 转化为类C代码, 以便阅读。我经过这次实验, 熟悉了IDA常用快捷键的使用, 以及各个窗口的切换等。另外, 我还了解了IDA Python, 学会在IDA中使用Python脚本辅助分析, 以达到事半功倍的效果。