

南開大學

恶意代码分析与防治技术实验报告

Lab5



学院：网络空间安全学院

专业：信息安全、法学

学号：2113203

姓名：付政烨

班级：信安法班

摘要

在“Lab 5-1”实验中，参与者进行了深入的恶意软件分析，探索了恶意软件的行为、工作原理和其背后的策略。该实验的核心是对一个恶意软件样本进行静态和动态分析，从而深入了解其内部结构、功能和行为。实验首先引导参与者使用 IDA Pro 等工具进行静态分析。这一步骤旨在让参与者直接查看恶意软件的代码结构，以识别关键的 API 调用、数据结构和和其他重要功能。这个阶段的挑战在于理解恶意软件的混淆和隐蔽策略，例如使用虚拟机检测来避免分析和编码数据来隐藏真实意图。接着，参与者进行了动态分析，实时观察恶意软件的运行行为。这不仅使参与者能够验证静态分析的发现，还提供了对恶意软件实际操作和其恶意活动的深入了解。例如，实验中揭示了恶意软件发出的 DNS 请求，以及它如何尝试与远程服务器通信。一个特别有趣的部分是对隐藏在恶意软件中的数据进行解码。这一步揭示了攻击者如何使用编码策略来隐蔽其意图，同时也展示了如何使用工具和技巧来揭示这些隐藏的信息。最后，实验还介绍了如何利用 Python 脚本来自自动化某些分析任务，例如解码恶意软件中的隐藏数据，这为参与者提供了一个实用的技能，可以在未来的恶意软件分析任务中使用。

关键字：恶意软件分析； 静态与动态分析； Ida Python

目录

一、 实验目的	1
二、 实验原理	1
三、 实验过程	1
(一) Lab 5-1	1
1. What is the address of DllMain?	1
2. Use the Imports window to browse to gethostbyname. Where is the import located?	2
3. How many functions call gethostbyname?	2
4. Focusing on the call to gethostbyname located at 0x10001757, can you figure out which DNS request will be made?	3
5. How many local variables has IDA Pro recognized for the subroutine at 0x10001656?	4
6. How many parameters has IDA Pro recognized for the subroutine at 0x10001656?	4
7. Use the Strings window to locate the string \cmd.exe /c in the disassembly. Where is it located?	4
8. What is happening in the area of code that references \cmd.exe /c?	5
9. In the same area, at 0x100101C8, it looks like dword __1008E5C4 is a global variable that helps decide which path to take. How does the malware set dword __1008E5C4? (Hint: Use dword __1008E5C4's cross-references.)	5
10. A few hundred lines into the subroutine at 0x1000FF58, a series of comparisons use memcmp to compare strings. What happens if the string comparison to robotwork is successful (when memcmp returns 0)?	6
11. What does the export PSLIST do?	9
12. Use the graph mode to graph the cross-references from sub __10004E79. Which API functions could be called by entering this function? Based on the API functions alone, what could you rename this function?	11
13. How many Windows API functions does DllMain call directly? How many at a depth of 2?	14
14. At 0x10001358, there is a call to Sleep (an API function that takes one parameter containing the number of milliseconds to sleep). Looking backward through the code, how long will the program sleep if this code executes?	15
15. At 0x10001701 is a call to socket. What are the three parameters?	15
16. Using the MSDN page for socket and the named symbolic constants functionality in IDA Pro, can you make the parameters more meaningful? What are the parameters after you apply changes?	16
17. Search for usage of the in instruction (opcode 0xED). This instruction is used with a magic string VMXh to perform VMware detection. Is that in use in this malware? Using the cross-references to the function that executes the in instruction, is there further evidence of VMware detection?	17
18. Jump your cursor to 0x1001D988. What do you find?	18

19.	If you have the IDA Python plug-in installed (included with the commercial version of IDA Pro), run Lab05-01.py, an IDA Pro Python script provided with the malware for this book. (Make sure the cursor is at 0x1001D988.) What happens after you run the script?	19
20.	With the cursor in the same location, how do you turn this data into a single ASCII string?	20
21.	Open the script with a text editor. How does it work?	20
四、 Yara 规则编写		20
五、 编写 IDA Python 脚本来辅助样本分析		21
(一)	课本样例分析	21
(二)	ida python 样例编写	22
六、 实验结论及心得体会		25

一、 实验目的

本次实验的目的是分析和解构一个名为”Lab 5-1”的恶意 DLL 文件。通过实验，我们将深入理解以下内容：如何在.text 和.idata 段中定位特定的函数和导入，如何识别和追踪特定 API 函数的调用，如何分析恶意软件中的字符串和其用途，如何识别和解释虚拟机检测技术，以及如何使用脚本对 IDA Pro 中的数据进行异或操作和修改。此外，实验还探讨了恶意软件如何与外部服务器通信、如何查询并发送注册表值、如何创建远程 shell 会话，以及如何识别和解码隐藏在恶意软件中的隐秘信息。

二、 实验原理

实验原理主要基于深入分析和解构恶意 DLL 文件的技术手段。其中，利用静态分析工具，如 IDA Pro，来查找和解释恶意软件中的关键功能、API 调用和数据段。这样可以揭示恶意软件的行为、它如何与外部服务器通信、以及如何获取和发送敏感信息。实验还涉及动态执行恶意代码以观察其运行时行为，从而更好地理解其操作和目的。此外，实验还探讨了恶意软件如何利用虚拟机检测技术来避免被分析，并通过特定的编码和解码技术隐藏其真实意图和信息。

三、 实验过程

(一) Lab 5-1

1. What is the address of DllMain?

使用 IDA Pro 工具来分析 lab05-01.dll 文件，首先打开该文件并导入进分析环境。接着，在工具的窗口中找到并点击“Functions”（函数）选项，以查找并定位到 DllMain 函数。根据分析结果，我们可以得知 DllMain 函数的地址为 0x1000D02E，如下图所示：

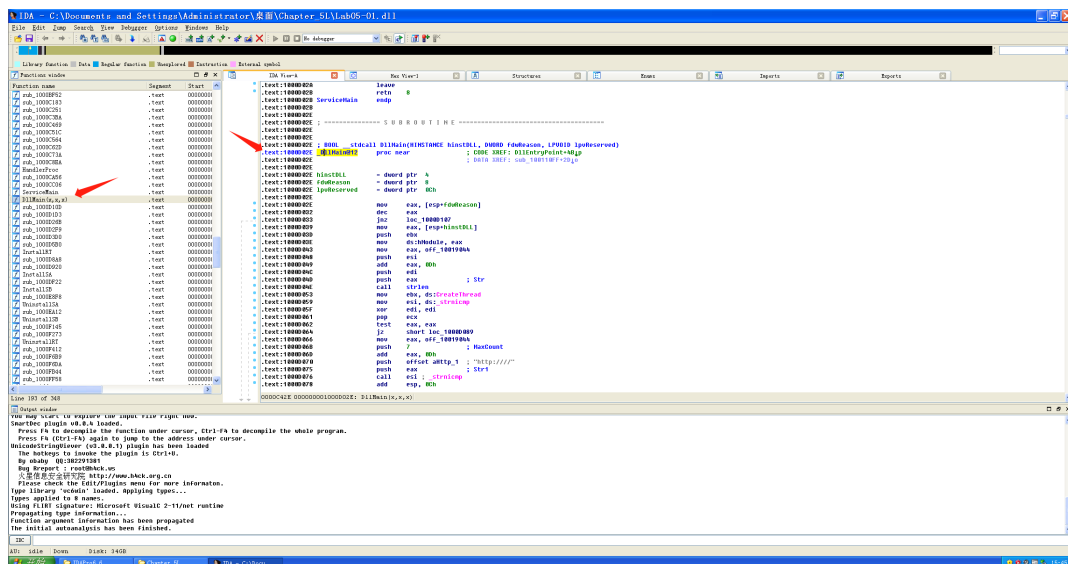


图 1

这一信息对于进一步的反汇编、逆向工程和漏洞分析等任务非常重要，因为它帮助我们定位并了解程序的入口点，以便更深入地分析和理解 lab05-01.dll 文件的内部结构和功能。

2. Use the Imports window to browse to gethostbyname. Where is the import located?

“gethostbyname”函数的导入位于.idata 部分的地址 0x100163CC 处。

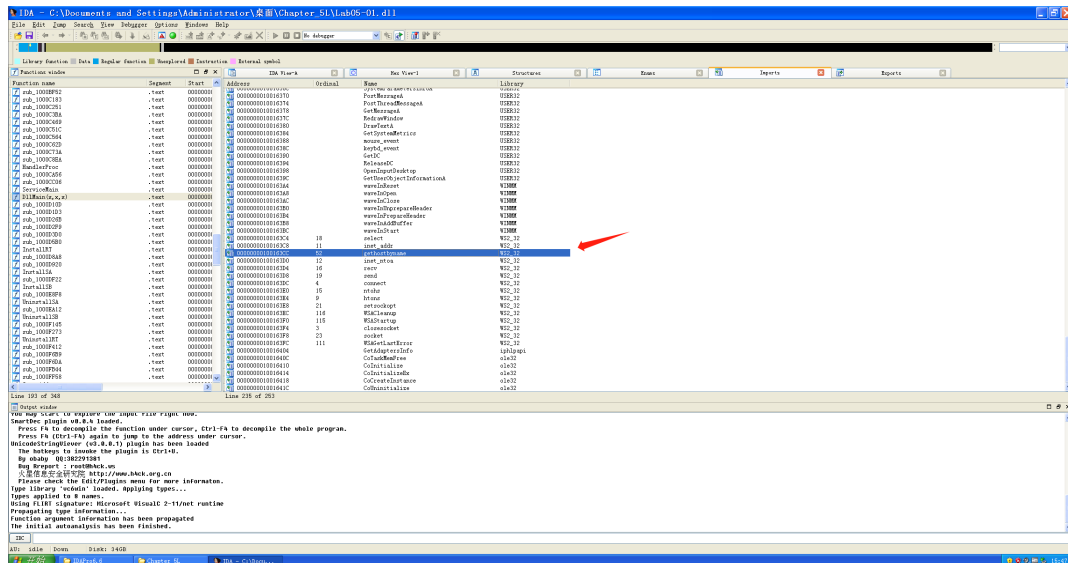


图 2

3. How many functions call gethostbyname?

在源代码中，我们首先定位了一个特定函数“gethostbyname”的代码片段，并使用快捷键 Ctrl+X 查看了该函数的交叉引用信息。

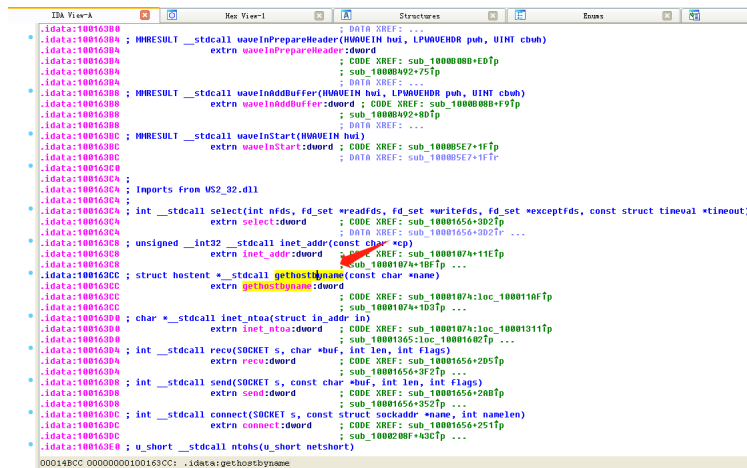


图 3

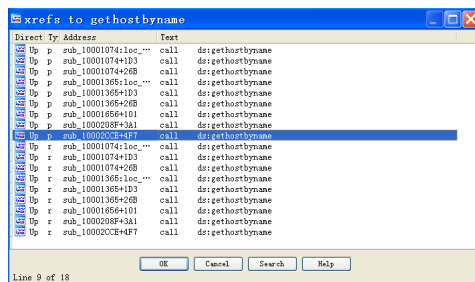


图 4

根据上述信息，我们可以了解到在代码中存在函数交叉引用的情况。在这个上下文中，我们使用了两个术语来描述这些引用：‘p’表示引用（即函数被其他部分引用），而‘r’表示读取（即函数被其他部分读取）。根据这些信息，我们可以得出结论，首先会进行函数的读取操作，然后才会进行引用操作。这一函数被引用了总共 9 次。此外，通过结合地址和偏移信息，我们可以确定这个函数被 5 个不同的函数调用。

4. Focusing on the call to `gethostbyname` located at `0x10001757`, can you figure out which DNS request will be made?

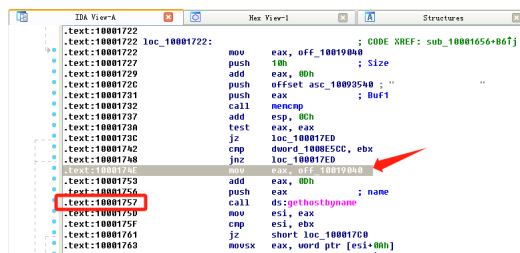


图 5

在应用程序的主界面中，通过按下键盘上的“G”键，实现了一个跳转操作，将执行流定位到内存地址 `0x10001757` 处。在这一位置，我们观察到了一段汇编代码，其首要任务是将内存地址 `0x10019040` 中的数据复制到寄存器 `eax` 中。为了更深入地了解这个过程，我们进一步分析了地址 `0x10019040` 处的代码块，通过双击“`off_10019040`”进入该地址。

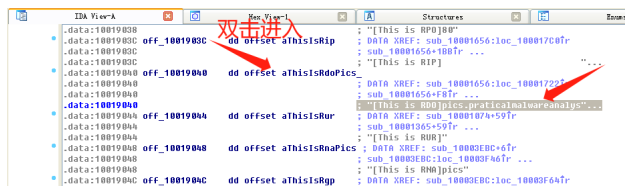


图 6

在地址 `0x10019040` 处，我们发现了字符串“`pics.practicalmalwareanalysis`”，并且通过进一步双击“`aThisIsRdoPics`”，我们发现了“`practicalmalwareanalysis.com`”的域名信息。这个过程表明，应用程序在这里可能正在准备与“`practicalmalwareanalysis.com`”通信的一些操作。

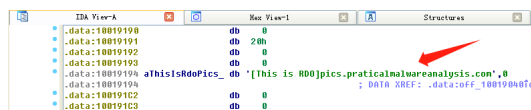


图 7

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 “practicalmalwareanalysis.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。



图 8

综上所述, 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. How many local variables has IDA Pro recognized for the subroutine at 0x10001656?

IDA Pro 的搜索结果显示, 识别了在 0x10001656 处的子过程中的 23 个局部变量。

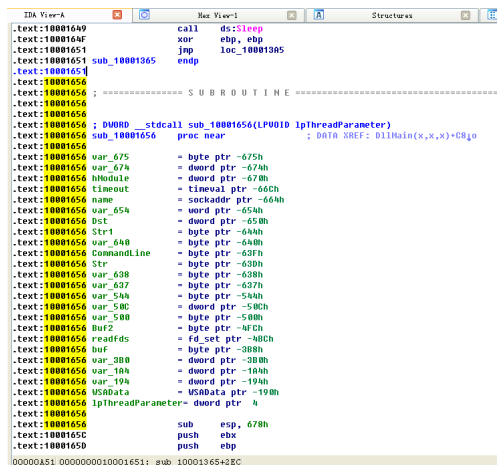


图 9

6. How many parameters has IDA Pro recognized for the subroutine at 0x10001656?

根据问题 5 中的图片, 我们观察到仅存在 1 个名为 “arg_0” 的参数。

7. Use the Strings window to locate the string \cmd.exe /c in the disassembly. Where is it located?

根据图示信息, 该字符串位于内存地址 0x10095B34 处

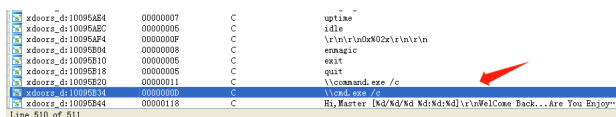


图 10



图 11

8. What is happening in the area of code that references \cmd.exe /c?

在深入研究所提供的代码片段并结合问题 7 的相关图表时，我们可以观察到在该特定代码区域存在着明显的交叉引用。通过双击以进入交叉引用区域，我们可以进一步深入分析。

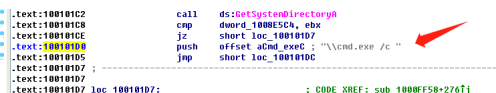


图 12

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。这一行为表明了程序执行过程中的某一时刻，该函数将被调用。

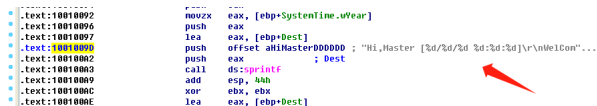


图 13

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“aHiMasterD-DDDDD”。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。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。

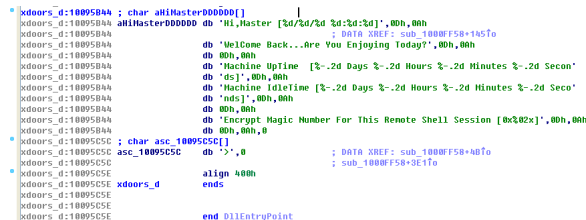


图 14

9. In the same area, at 0x100101C8, it looks like dword __ 1008E5C4 is a global variable that helps decide which path to take. How does the malware set dword __ 1008E5C4? (Hint: Use dword __ 1008E5C4' s cross-references.)

通过对以下代码段的分析，我们可以了解其功能和执行过程：

```

.text:10010100      push     eax                ; lpBuffer
.text:10010101      mov     [ebp+StartupInfo.dwFlags], 10h
.text:10010102      call    ds:GetSystemDirectory
.text:10010103      cmp     dword_1008E5C4, ebx
.text:10010104      jz      short loc_10010107
.text:10010105      push    offset aCmd_exe    ; "\\cmd.exe /c "
.text:10010106      jmp     short loc_1001010C
.text:10010107 loc_10010107:
.text:10010107      push    offset aCommand_exe ; "\\command.exe /c "

```

图 15

在程序中,我们首先搜索并进入了一个特定区域的代码段。接着,我们对地址 dword_1008E5C4 所包含的内容进行了详细分析。这个地址似乎与某些操作相关联。

```

.data:1008E5C4 dword_1008E5C4 dd ?
.data:1008E5C4 dword_1008E5C4 dd ?
.data:1008E5C4 dword_1008E5C4 dd ?
.data:1008E5C4 dword_1008E5C4 dd ?
.data:1008E5C4 dword_1008E5C4 dd ?
.data:1008E5C4 dword_1008E5C4 dd ?

```

图 16

通过使用 Ctrl+X 功能,我们查看了与这个地址相关的交叉引用。结果显示,只有一个引用是与写操作相关的,这引发了我们进一步的关注。

```

xrefs to dword_1008E5C4
Direct By Address Text
Up r sub_10007312+4B cmp dword_1008E5C4, edi
Up r sub_1000FF58+270 cmp dword_1008E5C4, ebx

```

图 17

在深入分析该特定区域的代码时,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:1000166F      mov     [esp+68h+hModule], ebx
.text:10001670      call   sub_10003695
.text:10001671      mov     dword_1008E5C4, eax
.text:10001672      call   sub_10003695
.text:10001673      push    3A9h                ; dwMilliseconds
.text:10001674      mov     dword_1008E5C8, eax
.text:10001675      call   ds:imp_77D110F
.text:10001676      call   sub_100110FF
.text:10001677      lea     eax, [esp+68h+USData]
.text:10001678      push    eax                  ; lpCmdData
.text:10001679      push    20h                 ; wVersionRequested
.text:10001680      call   ds:USStartup

```

图 18

为了理解 eax 寄存器的值是如何计算得出的,我们双击了前面的函数调用 sub_10003695。通过跳转到相关区域,我们了解到该函数包括一个对 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,以确定 al 寄存器的值,即 eax 寄存器的值。需要指出的是,值 2 代表 WIN32_NT 系统,这意味着该函数用于检查操作系统是否为 Win32 系统,并根据结果来设置 eax 的值。最终,获取的操作系统版本号会存储在 dword_1008E5C4 中,这个地址的值将是 0 或 1,反映了操作系统的类型。

10. A few hundred lines into the subroutine at 0x1000FF58, a series of comparisons use memcmp to compare strings. What happens if the string comparison to robotwork is successful (when memcmp returns 0)?

在分析特定区域的过程中,我们需要遵循一系列步骤以深入了解其功能和操作。此分析旨在解释在给定软件或系统中执行的特定操作序列,尤其是与注册表操作相关的情况。首先,我们要确定感兴趣的区域,通常通过程序的二进制可执行文件进行查找和定位。在此之后,我们可以通过特定的窗口或界面进一步查看区域内的数据和代码。

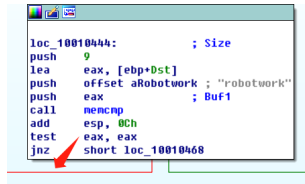


图 19

在成功定位目标区域后，通常需要遵循代码中的执行路径以深入理解其功能。这可以通过跟随代码中的控制流路径来实现，通常使用调试器或反汇编工具来支持这一过程。在这一示例中，我们跟随了一条红色箭头指示的路径，导致我们进入了 sub_100052A2 区域。

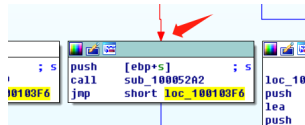


图 20

在 sub_100052A2 区域，我们开始查找与注册表相关的信息。注册表是 Windows 操作系统中用于存储配置和设置的重要数据库。我们可以合理地推断，sub_100052A2 区域可能包含与注册表操作相关的代码。

```
; int __cdecl sub_100052A2(SOCKET s)
sub_100052A2 proc near
Dest= byte ptr -60Ch
var_600= byte ptr -600h
Data= byte ptr -20Ch
var_200= byte ptr -200h
cbData= dword ptr -0Ch
Type= dword ptr -8
phkResult= dword ptr -4
s= dword ptr 8

push    ebp
mov     ebp, esp
sub     esp, 60Ch
call    [ebp+Dest], 0
push    edi
mov     ecx, 0FFh
xor     eax, eax
lea     edi, [ebp+var_600]
and     [ebp+Data], 0
rep stosd
stosb
push    7Fh
xor     eax, eax
pop     ecx
lea     edi, [ebp+var_200]
rep stosd
stosb
lea     eax, [ebp+phkResult]
push    eax                ; phkResult
push    0F003Fh            ; samDesired
push    0                  ; uiOptions
push    offset aSoftwareMicros ; "SOFTWARE\\Microsoft\\Windows\\CurrentVe..."
push    8000002h           ; hKey
call    ds:RegOpenKeyExA
test    eax, eax
jz      short loc_10005309
```

图 21

随后，我们继续向下追踪代码路径，最终到达了调用了 sub_100038EE 函数的位置。在这一步骤中，我们通过双击进入 sub_100038EE 函数区域，进一步深入研究其功能和操作。

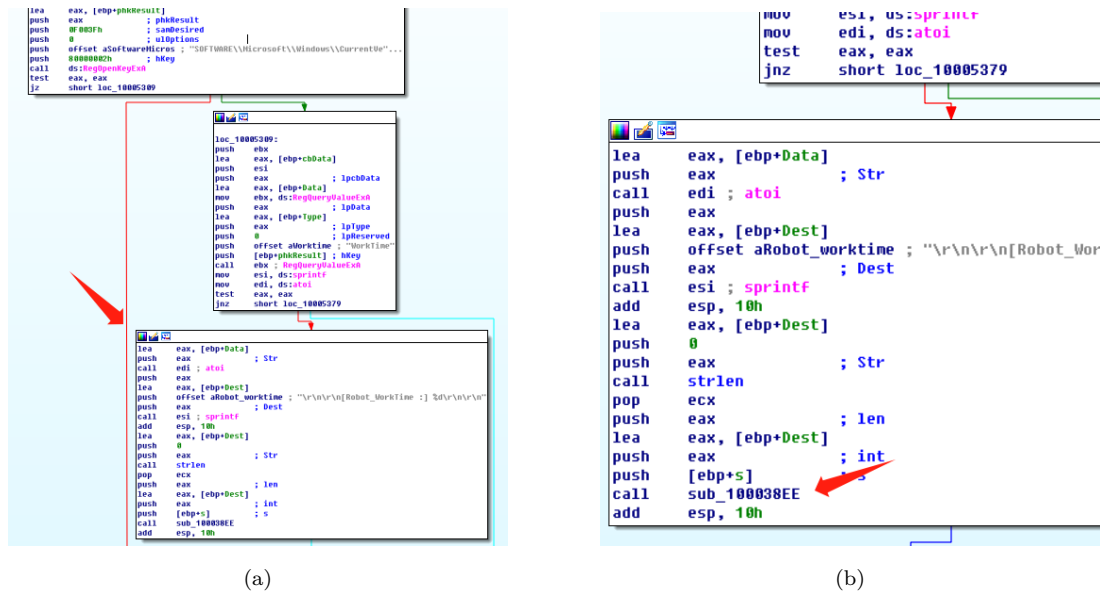


图 22

在 sub_100038EE 函数中，我们观察到它调用了 malloc 函数来分配内存空间，然后调用了 send 函数，最后调用了 free 函数来释放内存空间。这一序列操作表明，该函数可能涉及到一些与数据传输和内存管理相关的任务。

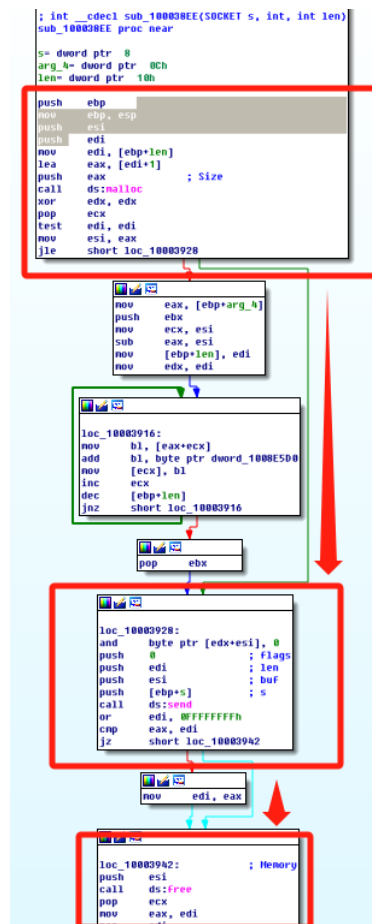


图 23

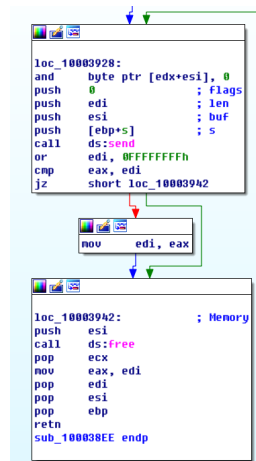


图 24

基于此观察，我们可以合理地推测，sub_100038EE 函数可能涉及对注册表路径"SOFTWARE\Microsoft\Windows\CurrentVersion" 的修改或查询操作，然后将相关结构发送出去。这种操作可能与配置信息或系统状态的获取或更改相关。

总之，通过深入的代码分析和路径追踪，我们可以初步推测出在给定区域中执行的操作可能涉及对 Windows 注册表的访问和与之相关的数据传输操作，这可能与系统配置或状态管理有关。然而，要进一步验证这些推测，需要进行更深入分析和验证。

11. What does the export PSLIST do?

进入导出表，我们对所讨论的函数的结构进行了深入研究。该函数的执行路径可以归纳如下：

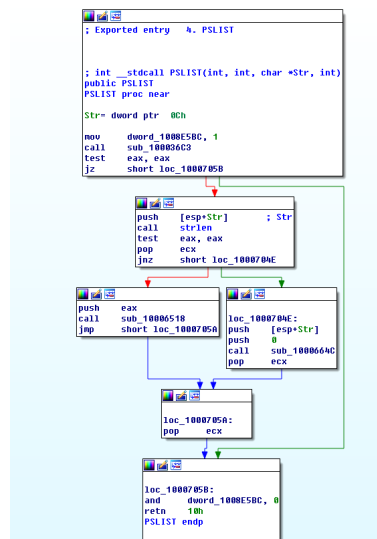


图 25

首先，该函数调用了子函数 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。



图 26

通过按 F5 查看伪 C 代码，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。

```

1  BOOL sub_100036C3()
2  {
3      struct _OSVERSIONINFOA VersionInformation; // [esp+0h] [ebp-94h]
4
5      VersionInformation.dwOSVersionInfoSize = 148;
6      GetVersionExA(&VersionInformation);
7      return VersionInformation.dwPlatformId == 2 && VersionInformation.dwMajorVersion
8          >= 5;
9  }

```

接下来，我们深入研究 sub_10006518 和 sub_1000664C 两个函数。

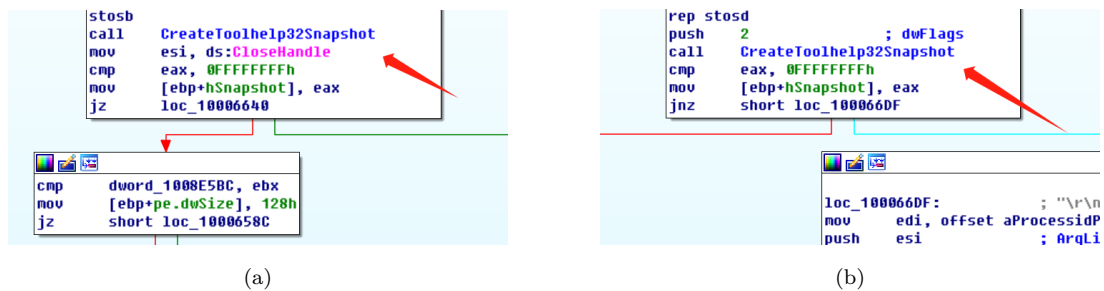


图 27

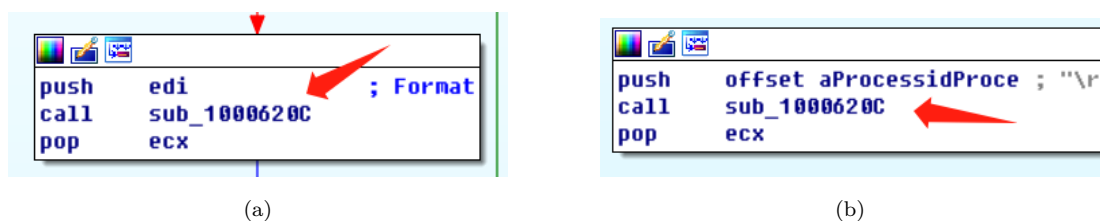


图 28

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。而 sub_1000620C 函数则是我们需要重点分析的部分。我们进一步深入研究 sub_1000620C 函数的内部逻辑：

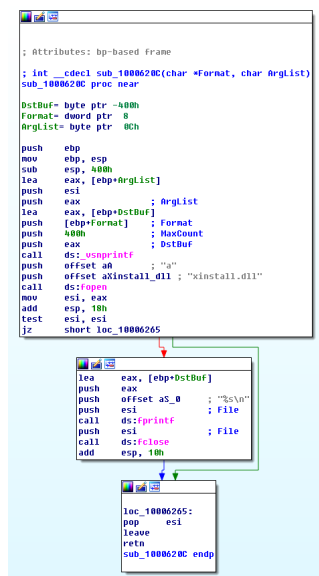


图 29

通过伪 C 代码的结合，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。

```

1 FILE *sub_1000620C(char *Format, ...)
2 {
3     FILE *result; // eax
4     FILE *v2; // esi
5     char DstBuf; // [esp+4h] [ebp-400h]
6     va_list va; // [esp+410h] [ebp+Ch]
7
8     va_start(va, Format);
9     vsnprintf(&DstBuf, 0x400u, Format, va);
10    result = fopen(aXinstallDll, aA);
11    v2 = result;
12    if ( result )
13    {
14        fprintf(result, aS_0, &DstBuf);
15        result = (FILE *)fclose(v2);
16    }
17    return result;
18 }
  
```

综上所述，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. Use the graph mode to graph the cross-references from sub _ 10004E79. Which API functions could be called by entering this function? Based on the API functions alone, what could you rename this function?

在程序分析过程中，我们识别出函数 sub_10004E79。

```
.text:10010438      jnz     short loc_10010444  
.text:1001043A      push    [ebp+s]  
.text:1001043D      call    sub_10004E79  
.text:10010442      jmp     short loc_100103F6  
.text:10010444
```

图 30

为了深入理解其上下文关系，我们选择此函数并生成其交叉引用图（User xrefs chart）。

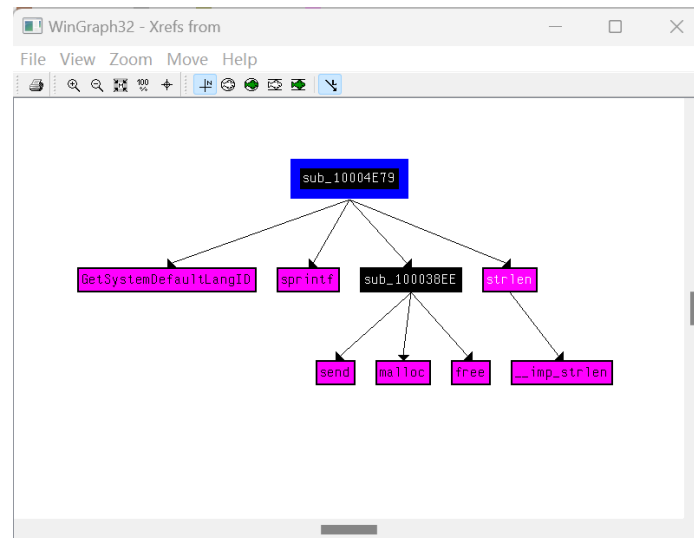


图 31

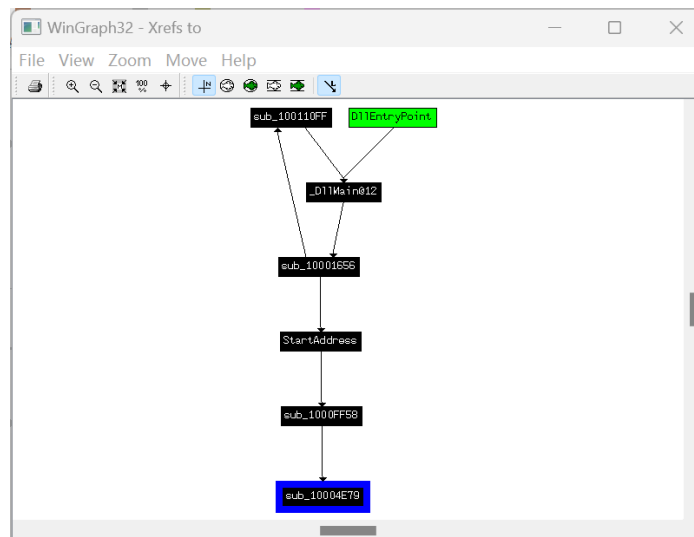


图 32

接着，对该函数进行进一步的代码分析。结果显示，当该函数被执行时，它会调用 API 函数：GetSystemDefaultLangID、sprintf 以及 strlen。此外，该函数还调用了另一个函数 sub_100038EE。深入分析 sub_100038EE，我们发现它进一步调用了以下函数：send、malloc、free 和 __imp_strlen。


```

.text:10004E79 ; int __cdecl sub_10004E79(SOCKET s)
.text:10004E79 sub_10004E79 proc near ; CODE XREF: sub_1000FF58+4E51p
.text:10004E79
.text:10004E79 Dest = byte ptr -400h
.text:10004E79 var_3FF = byte ptr -3FFh
.text:10004E79 s = dword ptr 8
.text:10004E79
.text:10004E79 push ebp
.text:10004E7A mov ebp, esp
.text:10004E7C sub esp, 400h
.text:10004E82 and [ebp+Dest], 0
.text:10004E89 push edi
.text:10004E8A mov ecx, 0FFh
.text:10004E8F xor eax, eax
.text:10004E91 lea edi, [ebp+var_3FF]
.text:10004E97 rep stosd
.text:10004E99 stosw
.text:10004E9B stosb
.text:10004E9C call ds:GetSystemDefaultLangID
.text:10004EA2 movzx eax, ax
.text:10004EA5 push eax
.text:10004EA6 lea eax, [ebp+Dest]
.text:10004EAC push offset aLanguageIDXX ; "\r\n\r\n[Language:] id:0x%x\r\n\r\n"
.text:10004EB1 push eax ; Dest
.text:10004EB8 call ds:sprintf
.text:10004EBB add esp, 0Ch
.text:10004EC1 lea eax, [ebp+Dest]
.text:10004EC3 push 0
.text:10004EC4 push eax ; Str
.text:10004EC4 call strlen
.text:10004EC9 pop ecx
.text:10004ECA push eax ; len
.text:10004ECB lea eax, [ebp+Dest]
.text:10004ED1 push eax ; int
.text:10004ED2 push [ebp+s]
.text:10004ED5 call sub_100038EE
.text:10004EDA add esp, 10h
.text:10004EDD pop edi
.text:10004EDE leave
.text:10004EDF retn
.text:10004EDF sub_10004E79 endp

```

图 33

```

.text:100038FA call ds:malloc
.text:10003900 xor edx, edx
.text:10003902 pop ecx
.text:10003903 test edi, edi
.text:10003905 mov esi, eax
.text:10003907 jle short loc_10003928
.text:10003909 mov eax, [ebp+arg_4]
.text:1000390C push ebx
.text:1000390D mov ecx, esi
.text:1000390F sub ebx, ecx
.text:10003911 mov [ebp+len], edi
.text:10003914 mov edx, edi
.text:10003916
.text:10003916 loc_10003916: mov bl, [eax+ecx] ; CODE XREF: sub_100038EE+371j
.text:10003919 add bl, byte ptr dword_1008E5D0
.text:1000391F mov [ecx], bl
.text:10003921 inc ecx
.text:10003922 dec [ebp+len]
.text:10003925 jnz short loc_10003916
.text:10003927 pop ebx
.text:10003928
.text:10003928 loc_10003928: and byte ptr [edx+esi], 0 ; CODE XREF: sub_100038EE+191j
.text:1000392C push 0 ; flags
.text:1000392E push edi ; len
.text:1000392F push esi ; buf
.text:10003930 push [ebp+s] ; s
.text:10003933 call ds:send
.text:10003935 or edi, 0FFFFFFFh
.text:1000393C cmp eax, edi
.text:1000393E jz short loc_10003942
.text:10003940 mov edi, eax
.text:10003942 loc_10003942: push esi ; CODE XREF: sub_100038EE+501j
.text:10003943 call ds:free ; Memory
.text:10003945 pop ecx
.text:10003947 mov eax, edi

```

图 34

```

.text:10014F4C jmp ds:__imp_strlen
.text:10014F4C strlen endp
.text:10014F4C
.text:10014F52 ; [00000006 BYTES: COLLAPSED FUNCTION memset. PRESS CTRL-NUMPAD+ TO EXPAND]
.text:10014F58 ; [00000006 BYTES: COLLAPSED FUNCTION memcmp. PRESS CTRL-NUMPAD+ TO EXPAND]
.text:10014F5E ; [00000006 BYTES: COLLAPSED FUNCTION printf. PRESS CTRL-NUMPAD+ TO EXPAND]

```

图 35

考虑到 sub_10004E79 直接调用了 GetSystemDefaultLangID 并且通过 sub_100038EE 间接调用了 send，我们推测该函数可能在获取系统的默认语言 ID 后，利用 send 函数将此 ID 发送至远程终端。基于这个行为特征，为该函数提议命名为 GetSystemLanguage。

13. How many Windows API functions does DllMain call directly? How many at a depth of 2?

在"Functions calls" 窗口中, 明确观察到以下情况:

Address	Called function
1 .text:1000D04E	call strlen
2 .text:1000D076	call esi; strcmp
3 .text:1000D092	call strlen
4 .text:1000D0AC	call esi; strcmp
5 .text:1000D0BF	call ebx; CreateThread
6 .text:1000D0DC	call esi; strcmp
7 .text:1000D0EE	call esi; strcmp
8 .text:1000D0FD	call ebx; CreateThread

图 36

该环境直接调用了四个 Windows API: strlen、strcmp、CreateThread 和 strcmp。当进入"User xrefs chart" 并取消选中"Cross" 和"references to" 选项, 同时设置深度为 2 时, 由于显示范围的限制, 相关的图形未予展示。然而, 从所得图像中可以分析出, 在深度为 2 的情况下, 大约有超过三十个 API 被调用, 例如 CreateThread 和 ExitThread 等。

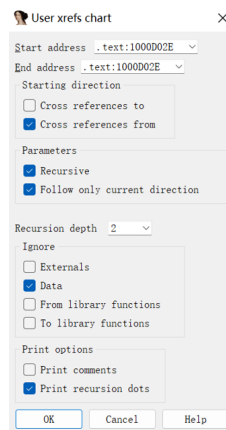


图 37

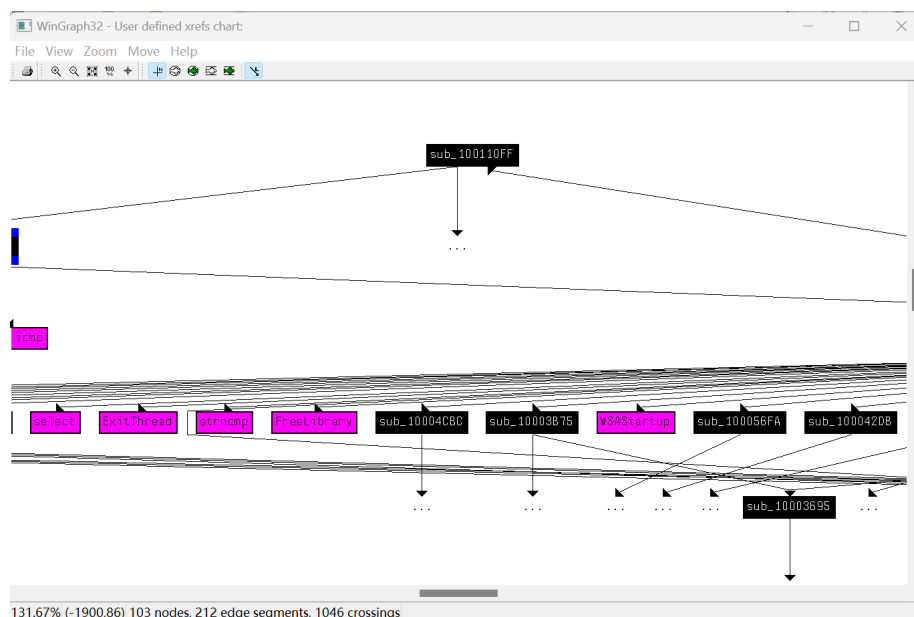
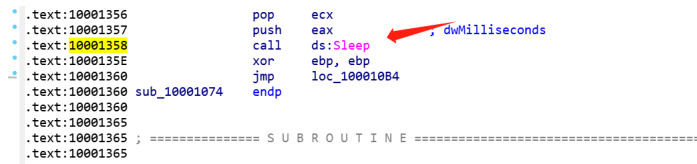


图 38

14. At 0x10001358, there is a call to Sleep (an API function that takes one parameter containing the number of milliseconds to sleep). Looking backward through the code, how long will the program sleep if this code executes?

对指定区域进行搜索：



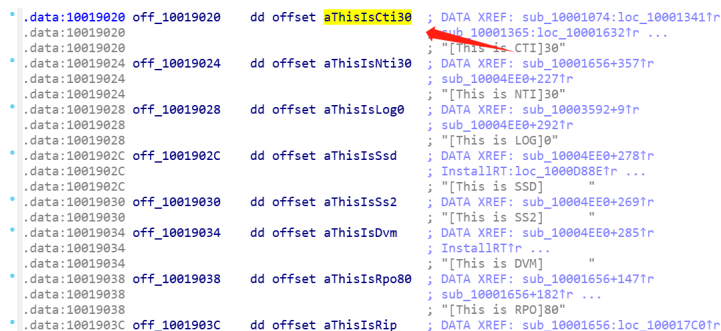
```

.text:10001356      pop     ecx
.text:10001357      push   eax
.text:10001358      call   ds:Sleep
.text:1000135E      xor     ebp, ebp
.text:10001360      jmp     loc_100010B4
.text:10001360      sub_10001074 endp

```

图 39

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。为了查看此地址处的内容，必须导航到 off_10019020。



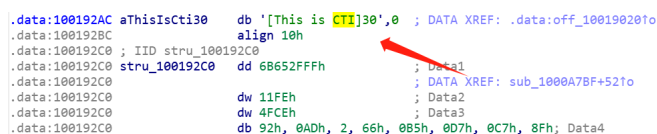
```

.data:10019020 off_10019020 dd offset aThisIsCti30 ; DATA XREF: sub_10001074:loc_10001341tr
.data:10019020      ; sub_10001365:loc_10001632tr ...
.data:10019020      ; "[This is CTI]30"
.data:10019024 off_10019024 dd offset aThisIsNti30 ; DATA XREF: sub_10001656+357tr
.data:10019024      ; sub_10004EE0+227tr
.data:10019024      ; "[This is NTI]30"
.data:10019028 off_10019028 dd offset aThisIsLog0 ; DATA XREF: sub_10003592+91tr
.data:10019028      ; sub_10004EE0+292tr
.data:10019028      ; "[This is LOG]0"
.data:1001902C off_1001902C dd offset aThisIsSsd ; DATA XREF: sub_10004EE0+278tr
.data:1001902C      ; InstallRT:loc_1000D8E1tr ...
.data:1001902C      ; "[This is SSD]"
.data:10019030 off_10019030 dd offset aThisIsSs2 ; DATA XREF: sub_10004EE0+269tr
.data:10019030      ; "[This is SS2]"
.data:10019034 off_10019034 dd offset aThisIsDvm ; DATA XREF: sub_10004EE0+285tr
.data:10019034      ; InstallRT1tr ...
.data:10019034      ; "[This is DVM]"
.data:10019038 off_10019038 dd offset aThisIsRpo80 ; DATA XREF: sub_10001656+147tr
.data:10019038      ; sub_10001656+182tr ...
.data:10019038      ; "[This is RPO]80"
.data:1001903C off_1001903C dd offset aThisIsRip ; DATA XREF: sub_10001656:loc_100017C0tr

```

图 40

在该地址，我们观察到了 [This is CTI]30 的字符串。进一步导航到 aThisIsCti30，我们确认其定义的字符串为 [This is CTI]30。



```

.data:100192AC aThisIsCti30 db '[This is CTI]30',0 ; DATA XREF: .data:off_10019020fo
.data:100192BC      align 10h
.data:100192C0 ; IID stru_100192C0
.data:100192C0 stru_100192C0 dd 6B652FFFh ; Data1
.data:100192C0      ; Data2
.data:100192C0      ; Data3
.data:100192C0      ; Data4

```

图 41

回到调用 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. At 0x10001701 is a call to socket. What are the three parameters?

通过代码区域的深入分析：

```

.text:100016F8      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

```

图 42

依据栈帧的结构化布局，我们可以推断出三个参数的值分别为 6、1 和 2。

16. Using the MSDN page for socket and the named symbolic constants functionality in IDA Pro, can you make the parameters more meaningful? What are the parameters after you apply changes?

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。

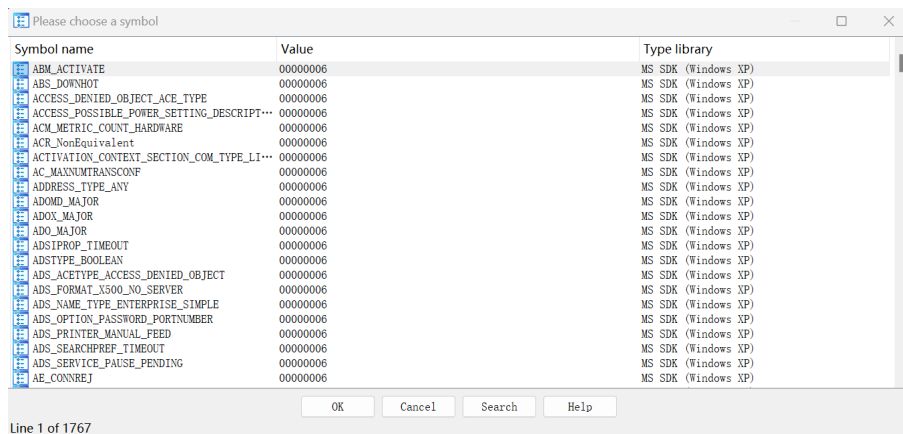


图 43

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，protocol 值为 6 对应于 IPPROTO_TCP，即采用 TCP 作为传输协议。

IPPROTO_TCP 6	The Transmission Control Protocol (TCP). This is a possible value when the <i>af</i> parameter is AF_INET or AF_INET6 and the <i>type</i> parameter is SOCK_STREAM.
SOCK_STREAM 1	A socket type that provides sequenced, reliable, two-way, connection-based byte streams with an OOB data transmission mechanism. This socket type uses the Transmission Control Protocol (TCP) for the Internet address family (AF_INET or AF_INET6).
AF_INET	The Internet Protocol version 4 (IPv4) address family.

type的1代表使用internet地址系列，指要使用IP进行通信。

af的2代表使用IPv4进行通信。

图 44

应用了修改后参数如下：

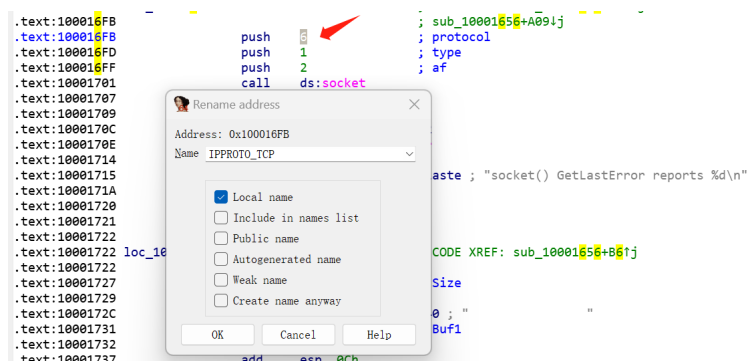


图 45



图 46

17. Search for usage of the `in` instruction (opcode `0xED`). This instruction is used with a magic string `VMXh` to perform VMware detection. Is that `in` in use in this malware? Using the cross-references to the function that executes the `in` instruction, is there further evidence of VMware detection?

在对 `in` 进行深入分析后，我们识别到了特定的代码段。

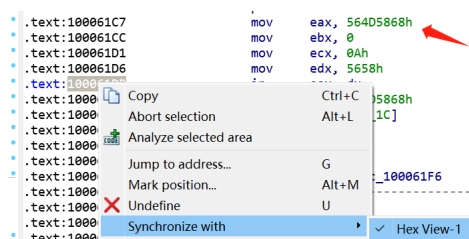


图 47

在定位到 `in` 指令的具体地址后，观察到一系列类似于字符串的十六进制数值。

```

.text:100061C7      mov     eax, 'VMXh'
.text:100061CC      mov     ebx, 0
.text:100061D1      mov     ecx, 0Ah
.text:100061D6      mov     edx, 'VX'
.text:100061DB      in      eax, dx
.text:100061DC      cmp     ebx, 'VMXh'

```

图 48

对这些十六进制数值进行选择，并采用 R 键进行解码，其解析结果为“VMXh”。这表明相关代码部分应用了 Vmware 的检测机制。

进一步探索该指令相关的函数交叉引用。

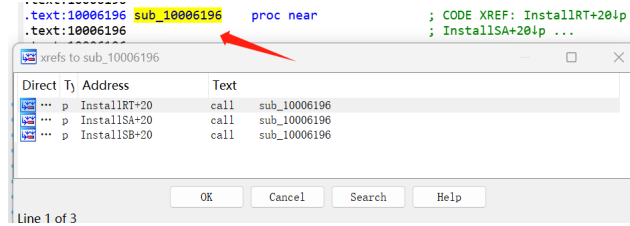


图 49

细致地检查每个关联函数后，

```

.text:1000D870 loc_1000D870:      ; CODE XREF: InstallRT+1E1j
.text:1000D870      push    offset unk_1008E5F0 ; Format
.text:1000D875      call    sub_10003592
.text:1000D87A      mov     [esp+8+Format], offset aFoundVirtualMa ; "Found Virtual Machine,Install Cancel."
.text:1000D881      call    sub_10003592
.text:1000D886      pop     ecx
.text:1000D887      call    sub_10005567
.text:1000D88C      jmp     short loc_1000D8A4

```

图 50

```

.text:1000DEEA loc_1000DEEA:      ; CODE XREF: InstallSA+1E1j
.text:1000DEEA      push    offset unk_1008E5F0 ; Format
.text:1000DEEF      call    sub_10003592
.text:1000DEF4      mov     [esp+8+Format], offset aFoundVirtualMa ; "Found Virtual Machine,Install Cancel."
.text:1000DEFB      call    sub_10003592
.text:1000DF00      pop     ecx
.text:1000DF01      call    sub_10005567
.text:1000DF06      jmp     short loc_1000DF1E

```

图 51

```

.text:1000E8BB loc_1000E8BB:      ; CODE XREF: InstallSB+1E1j
.text:1000E8BB      push    offset unk_1008E5F0 ; Format
.text:1000E8C0      call    sub_10003592
.text:1000E8C5      mov     [esp+8+Format], offset aFoundVirtualMa ; "Found Virtual Machine,Install Cancel."
.text:1000E8CC      call    sub_10003592
.text:1000E8D1      pop     ecx
.text:1000E8D2      call    sub_10005567
.text:1000E8D7      jmp     short loc_1000E8F4

```

图 52

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。

18. Jump your cursor to 0x1001D988. What do you find?

如图所示，我们观察到了一系列随机数据。

```

.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 ; &
.data:1001D9A1 db 21h ; !
.data:1001D9A2 db 27h ; '
.data:1001D9A3 db 3Ch ; <

```

图 53

19. If you have the IDA Python plug-in installed (included with the commercial version of IDA Pro), run Lab05-01.py, an IDA Pro Python script provided with the malware for this book. (Make sure the cursor is at 0x1001D988.) What happens after you run the script?

在执行“run script”操作后，这些随机数据被转化为明文形式。

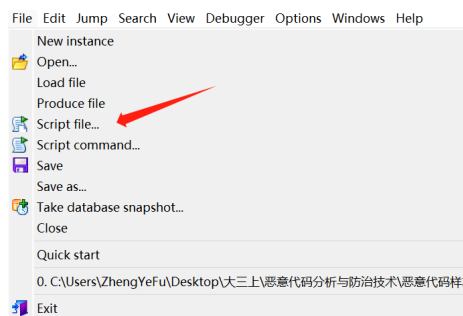


图 54

```

.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 55

20. With the cursor in the same location, how do you turn this data into a single ASCII string?

当按下“A” 键时，我们获得以下输出：

```
.data:1001D988 aXdoorIsThisBac db 'xdoor is this backdoor, string decoded for Practical Malware Anal'
.data:1001D988 db 'ysis Lab :)1234',0
.data:1001D9D9 db 0
.data:1001D9DA db 0
.data:1001D9DB db 0
.. .. ..
```




图 56

这是一个字符串，其内容为“xdoor is this backdoor, string decoded for Practical Malware Analysis Lab :)1234”。

21. Open the script with a text editor. How does it work?

‘ScreenEA()’ 函数用于检索 IDA 调试窗口中由光标指示的代码的地址。在其后，一个 ‘for’ 循环被设计来遍历连续的 50 个字节。在此循环中，使用 ‘Byte’ 函数来提取每个字节的具体值。此值随后与 0x55 进行异或 (‘XOR’) 运算。最终，运算的结果被写回到 IDA 中与原始地址对应的位置。

四、 Yara 规则编写

```
1 rule Rule_Lab05_01 {
2     meta:
3         description = "Lab05-01.dll"
4
5     strings:
6         $injectQuery = "Maybe Inject Mode?" nocase fullword ascii
7         $uninjectNotFound = "Process '%s' Not Found,Uninject Failed" fullword ascii
8         $uninjectFromProcess = "Uninject '%s' From Process '%s' " fullword ascii
9         $processInfoError = "error on get process info. " fullword ascii
10        $injectNotFound = "Process '%s' Not Found ,Inject Failed" fullword ascii
11        $cmdExePath = "\\cmd.exe /c " fullword ascii
12        $injectFailMessage = "Inject '%s' To Process '%s' Failed" fullword ascii
13        $xfloodDll = "xflood.dll" fullword ascii
14        $injectToProcess = "Inject '%s' To' %x' Process '%s'" fullword ascii
15        $commandExePath = "\\command.exe /c " fullword ascii
16        $uninjectFromProcessAlt = "Uninject '%s' From Process '%s'" fullword ascii
17        $rundll64Exe = "rundll64.exe" fullword ascii
18        $rundll32ExeCommand = "rundll32.exe %s,StartEXS %s:%s" fullword ascii
19        $xproxyDll = "xproxy.dll" fullword ascii
20        $xkeyDll = "xkey.dll" fullword ascii
21        $xsysDll = "xsys.dll" fullword ascii
22        $xinstallDll = "xinstall.dll" fullword ascii
23        $xdevDll = "xdev.dll" fullword ascii
24
25    condition:
26        uint16(0) == 0x5a4d and
27        uint32(uint32(0x3c)) == 0x00004550 and
28        filesize < 400KB and
29        1 of ($injectQuery, $uninjectNotFound, $uninjectFromProcess,
30            $processInfoError, $injectNotFound, $cmdExePath, $injectFailMessage) and
```



```

30     5 of them
31 }

```

该 YARA 规则旨在在文件中搜索特定的字符串模式，以识别与“Lab05-01.dll”相关的恶意活动或特征。同时，如果文件中的内容满足规则中定义的条件，如：文件的开头两个字节（通常称为“魔数”）必须是 0x5a4d，这是 PE（可执行）文件的典型标识；在由文件中的 0x3c 偏移量指定的位置，四个字节的价值必须为 0x00004550，进一步验证 PE 文件格式；文件的大小必须小于 400KB；文件中至少需要存在上述定义的一个字符串模式；文件中必须至少存在上述定义的五条字符串模式。则认为文件与该规则匹配，扫描结果如下：

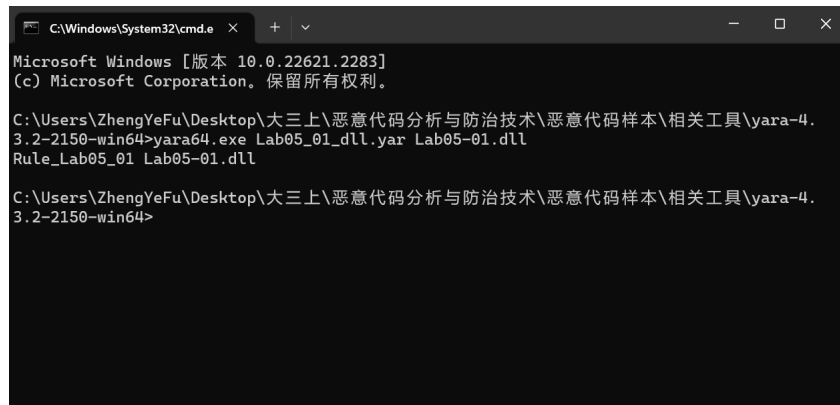


图 57

五、 编写 IDA Python 脚本来辅助样本分析

（一） 课本样例分析

首先，我们对课本中给出的样例进行详细的分析：

```

1  sea = ScreenEA()
2
3  for i in range(0x00,0x50):
4      b = Byte(sea+i)
5      decoded_byte = b ^ 0x55
6      PatchByte(sea+i,decoded_byte)

```

上述代码的具体功能与实现原理如下：

- sea = ScreenEA() 这一行代码调用了 ScreenEA 函数，该函数在 IDA Pro 中用于获取当前在调试窗口中光标所指向代码的起始地址。得到的地址被保存在变量 sea 中。
- for i in range(0x00,0x50): 这是一个循环，范围从 0x00 到 0x4F（十进制中的 0 到 79），总共 80 步，但由于 Python 的 range 函数不包括结束值，所以实际上是从 0 到 79，即总共 80 个字节。
- b = Byte(sea+i) 这一行代码获取了从起始地址 sea 开始，偏移 i 个字节位置的字节值。该值被保存在变量 b 中。
- decoded_byte = b ^ 0x55 这行代码对上一步中获取的字节值 b 进行异或（XOR）操作。这里的异或操作是一个简单的加密或解密技术，它将字节值 b 与 0x55 进行异或。由于异或

的性质（相同的值异或结果为 0，不同的值异或结果为 1），当同一个值被异或两次时，会返回原始值。所以，如果原始字节已经与 0x55 异或过一次，这个操作会对其进行解码。

- PatchByte(sea+i,decoded_byte) 这行代码将解码后的字节值 decoded_byte 重新写入原始地址（即 sea + i 的位置）。在 IDA Pro 中，PatchByte 函数被用于修改一个给定地址的字节值

（二） ida python 样例编写

函数检测脚本

这段代码的主要功能是枚举和打印在 IDA Pro 中识别的所有函数的地址范围及其大小。它可以帮助他们快速地了解一个二进制文件中的函数布局，以及每个函数占用的空间大小。这可以为进一步的分析提供上下文，例如确定哪些函数可能包含更多的逻辑（因为它们更大）或者确定特定的代码模式。

列出所有函数及其大小

```
1 import idutils # 该模块包含了与IDA数据库交互的各种实用函数，
2 import idc # 使得脚本编写者可以与IDA的当前环境进行交互，
3
4 for func in idutils.Functions():
5     start = func
6     end = idc.find_func_end(func)
7     print("Function: 0x%x - 0x%x, Size: %d" % (start, end, end - start))
```

代码具体含义如下：

- for func in idutils.Functions(): 这一行开始一个循环，遍历 IDA 数据库中识别的每一个函数的起始地址。
- start = func: 存储当前函数的起始地址。
- end = idc.find_func_end(func): 使用 idc.find_func_end 函数来查询当前函数的结束地址。
- print("Function: 0x%x - 0x%x, Size: %d" % (start, end, end - start)): 打印当前函数的起始地址、结束地址以及函数的大小（即字节数）。

效果如下图所示：

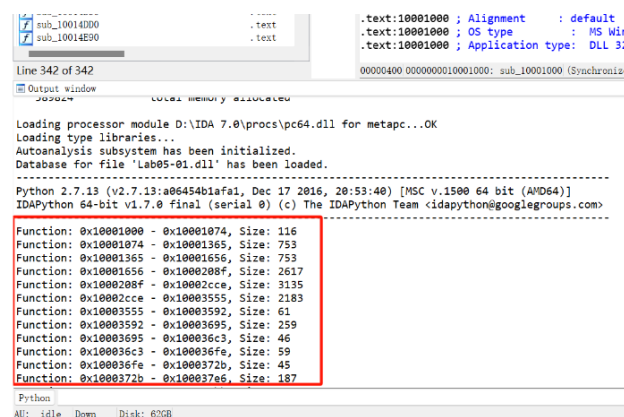


图 58

查找 API 调用脚本

该 IDA Python 脚本旨在找出在二进制文件中调用了特定 API（例如“CreateFileA”）的所有函数，并打印出这些函数和 API 调用的具体地址。由于手动在大型二进制文件中搜索特定的 API 调用可能非常耗时。这个脚本可以自动化这个过程，节省我们的时间。

查找特定类型的 API 调用

```

1 import idutils
2 import idc
3 import ida_funcs
4
5 def list_api_calls(api_name):
6     for addr, name in idutils.Names():
7         if api_name in name:
8             for xref in idutils.XrefsTo(addr):
9                 print("API call in function 0x%x at 0x%x" % (ida_funcs.get_func(
10                     xref.frm).start_ea, xref.frm))
11 list_api_calls("CreateFileA")

```

代码的具体实现原理如下：

- list_api_calls(api_name): 自定义函数，接受一个 API 名称（如“CreateFileA”）作为参数。
- 使用 idutils.Names() 遍历二进制文件中的所有命名地址，这通常包括函数、变量和导入的 API 名称。
- 使用条件语句 if api_name in name 来检查当前的名称是否包含指定的 API 名称。
- 如果找到匹配的 API 名称，该脚本使用 idutils.XrefsTo(addr) 来找出所有引用到该 API 地址的位置。
- 对于每个引用，该脚本会打印调用 API 的函数的起始地址和 API 调用的具体地址。

效果如下图所示：

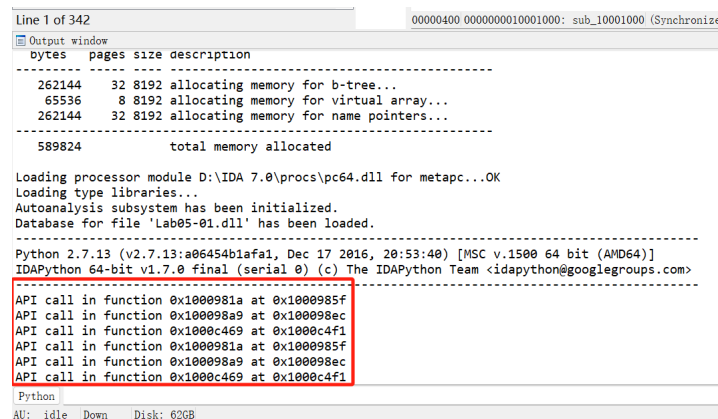


图 59

异或操作信息捕获脚本

XOR 是一种常用的简单的数据遮盖技术。恶意软件经常使用它来混淆其载荷、配置或与其 C&C 服务器通信的命令。如果恶意代码包含 XOR 操作，并且该操作的操作数之一是一个已知的或可猜测的值，那么可以尝试使用这个值来解密被 XOR 的数据。同时，如果一个函数被多次调用，这可能意味着它执行了某种核心功能或重要任务。在恶意代码中，这可能是与网络通信、数据加密、文件操作或其他恶意行为相关的函数。综合来看，脚本尝试找出那些可能用于混淆、遮盖真实行为或执行关键任务的函数。对于我们来说，这些函数是重要的分析目标，因为它们可能揭示了恶意软件的主要行为或其尝试隐藏的意图，下面编写的异或操作信息捕获脚本可以有效的帮助我们完成上述分析认为：

```
1  # coding=utf-8
2
3  import idautils
4  import idc
5
6  def find_xor_functions():
7      # 遍历所有函数
8      for func_ea in idautils.Functions():
9          # 为了检查函数是否包含XOR指令和被调用多次，我们需要设置两个计数器
10         xor_count = 0
11         call_count = 0
12
13         # 遍历函数中的每个指令
14         for ins_ea in idautils.FuncItems(func_ea):
15             mnem = idc.print_insn_mnem(ins_ea)
16             if mnem == "xor":
17                 xor_count += 1
18
19         # 遍历到该函数的所有交叉引用
20         for xref in idautils.XrefsTo(func_ea):
21             call_count += 1
22
23         # 检查是否满足条件：包含XOR并被调用多次
24         if xor_count > 0 and call_count > 1:
25             print("Function: %s at 0x%x contains XOR and is called %d times" % (idc
26                 .get_func_name(func_ea), func_ea, call_count))
27
28 find_xor_functions()
```

该脚本的目标是找出包含 XOR 指令并且被多次调用的函数，并在控制台打印这些函数的名称和地址。

1. idautils.Functions(): 遍历二进制文件中的所有函数。
2. idc.GetMnem(ea) == 'xor': 检查当前指令是否是 XOR。
3. idc.GetFunctionName(ea): 获取当前地址 ea 所在的函数名称。
4. len(list(idautils.CodeRefsTo(fea, 0))) > 1: 确保该函数被调用次数超过一次。
5. print("[+] Function %s at 0x%x contains XOR instruction and is called multiple times."
% (function_name, fea)): 在控制台打印满足条件的函数名称和地址。

效果如下图所示：

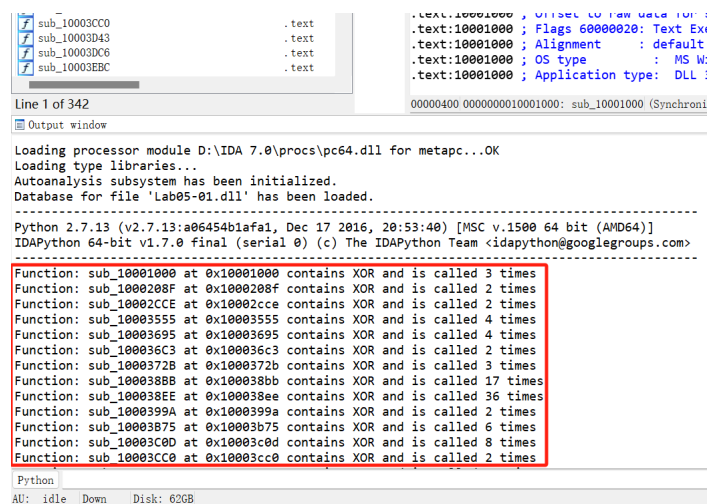


图 60

六、 实验结论及心得体会

在进行“Lab 5-1”实验的过程中，我深刻体会到了恶意软件分析的复杂性与挑战性。每一步的深入探索都像是解开一个谜题，使我更加了解恶意软件的工作原理和它们是如何被设计来达到其恶意目的的。

首先，使用 IDA Pro 等工具进行静态分析，我得以直接看到代码的结构和功能，这是一个极为宝贵的学习经验。它不仅帮助我识别关键的 API 调用和数据结构，还使我了解到恶意软件如何巧妙地隐藏其行为，如使用虚拟机检测技术和数据编码。其次，动态分析恶意软件的行为使我更加深入地了解了它们是如何实际运行的，以及它们试图达到的目标。这种实时观察为我提供了宝贵的直观感受，使我更好地理解静态分析的结果。此外，通过解码隐藏在恶意软件中的信息，我意识到了攻击者如何巧妙地设计策略来避免被检测和分析。这也强调了为何持续的学习和更新知识是如此重要，因为恶意软件的技术和策略总是在不断地进化。

同时，在这次的 IDAPython 脚本编写过程中，我深刻地体验到了自动化逆向工程分析的强大能力和效率。首先，使用 IDAPython 来扩展 IDA 的功能是一个极其有效的方式，它使得复杂的查询和操作变得简单化，节省了大量手动分析的时间。当面对大型或是结构复杂的二进制文件时，手动的分析和跟踪可能非常耗时和困难。但是，通过编写 IDAPython 脚本，我们可以轻松地找到特定的代码模式、函数调用或是特定的 API 引用，这大大加快了分析的速度。

参考文献

- [1] SM-D.Practical Malware Analysis[J].Network Security, 2012, 2012(12):4-4.DOI:10.1016/S1353-4858(12)70109-5.