

南开大学

恶意代码分析与防治技术实验报告

Lab05: IDA Python



学 院 网络空间安全学院
专 业 信息安全、法学
学 号 2112514
姓 名 辛浩然
班 级 信息安全、法学

一、实验目的

1. 使用 IDA PRO 分析病毒样本，熟悉 IDA PRO 的使用；
2. 编写 Yara 规则。
3. 利用 IDA Python 辅助病毒分析。

二、实验原理

IDA Pro 是一款强大的反汇编工具，它的功能远不止反汇编整个程序。除了反汇编，它还能执行各种其他任务，如查找函数、进行栈分析、标识本地变量等等。这使得 IDA Pro 成为逆向工程师和安全研究人员的利器。

IDA Pro 的一个独特特点是其快速库标记和识别技术(FLIRT)，它包含了可扩展的代码特征，可以识别并标记被反汇编的函数，尤其是编译器添加的库代码。这有助于用户更好地理解程序的结构和功能。

IDA Pro 是一款交互式工具，它允许用户修改、操作、重新安排或重新定义反汇编过程中的所有属性。这为逆向工程师提供了极大的灵活性，使他们能够更好地分析和理解程序的内部工作原理。

另一个重要的优势是 IDA Pro 具有保存分析过程的能力。用户可以添加注释、标记数据和函数名，并将这些工作保存到一个 IDA Pro 数据库，通常被称为 idb 文件。这样，用户可以在将来继续工作，而不必从头开始进行分析，从而提高了效率。

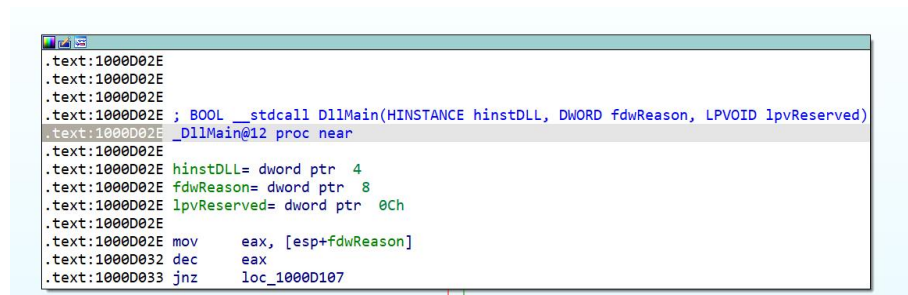
IDA Python 是一种 Python 编程接口和脚本工具，用于与 IDA Pro 集成，扩展和自动化 IDA Pro 反汇编器和逆向工程平台的功能。它为逆向工程师和安全研究人员提供了一种强大的方式来执行各种任务，包括自动化分析、自定义插件开发、脚本编写以及与 IDA Pro 进行交互。

三、实验过程

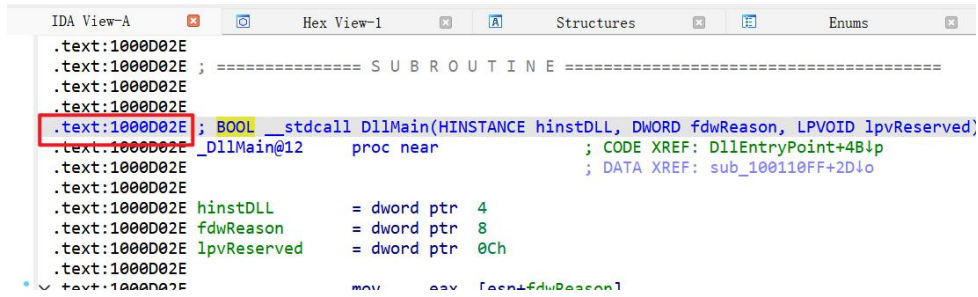
在 IDA PRO 中打开 Lab05-01.的 dll 文件，进行如下分析：

1. DllMain 的地址是什么？

用 IDA Pro 加载了恶意 DLL 以后，就直接来到了位于 0x1000D02E 处的 DllMain 函数。



```
.text:1000D02E
.text:1000D02E
.text:1000D02E
.text:1000D02E ; BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
.text:1000D02E _DllMain@12 proc near
.text:1000D02E
.text:1000D02E hinstDLL= dword ptr 4
.text:1000D02E fdwReason= dword ptr 8
.text:1000D02E lpvReserved= dword ptr 0Ch
.text:1000D02E
.text:1000D02E mov     eax, [esp+fdwReason]
.text:1000D032 dec     eax
.text:1000D033 jnz     loc_1000D107
```



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

在导入表中，找到 gethostbyname，它在 .idata 节的 0x100163CC 处。

Address	Ordinal	Name	Library
00000000100163CC	52	gethostbyname	WS2_32

.idata:100163C8		; Import by ordinal 11	
.idata:100163CC		; struct hostent *(__stdcall *gethostbyname)(const char *name)	
.idata:100163CC		extrn gethostbyname:dword	
.idata:100163CC		; CODE XREF: sub_10001074:loc_100011AFp	
.idata:100163CC		; sub_10001074+1D31p ...	
.idata:100163CC		; Import by ordinal 52	

3. 有多少函数调用了 gethostbyname？

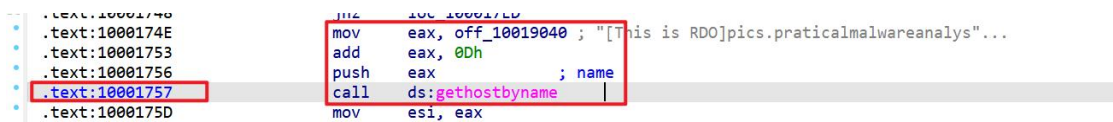
在 gethostbyname 处，按 Ctrl+X 键检查它的交叉引用情况。窗口中“Line 1 of 18”说明存在在对 gethostbyname 的 9 处交叉引用。因为 IDA Pro 会计算两次交叉引用：类型 p 是被调用的引用，类型 r 是被“读取”的引用(因为是对一个导入项 call dword ptr [...])，所以 CPU 必须先读取这个导入项，再调用它)。

仔细检查交叉引用列表，会发现实际上在 5 个不同的函数中调用了 gethostbyname。

Direction	Type	Address	Text
Up	r	sub_10001074:loc_100011AF	call ds:gethostbyname
Up	p	sub_10001074:loc_100011AF	call ds:gethostbyname
Up	p	sub_10001074+1D3	call ds:gethostbyname
Up	p	sub_10001074+1D3	call ds:gethostbyname
Up	r	sub_10001074+26B	call ds:gethostbyname
Up	p	sub_10001074+26B	call ds:gethostbyname
Up	r	sub_10001365:loc_100014A0	call ds:gethostbyname
Up	p	sub_10001365:loc_100014A0	call ds:gethostbyname
Up	r	sub_10001365+1D3	call ds:gethostbyname
Up	p	sub_10001365+1D3	call ds:gethostbyname
Up	r	sub_10001365+26B	call ds:gethostbyname
Up	p	sub_10001365+26B	call ds:gethostbyname
Up	r	sub_10001656+101	call ds:gethostbyname
Up	p	sub_10001656+101	call ds:gethostbyname
Up	r	sub_1000208F+3A1	call ds:gethostbyname
Up	p	sub_1000208F+3A1	call ds:gethostbyname
Up	r	sub_10002CCE+4F7	call ds:gethostbyname
Up	p	sub_10002CCE+4F7	call ds:gethostbyname

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

查看调用出代码：



其中，因为 off_10019040+0Dh 被赋给了 EAX，所以 gethostbyname 方法用了一个参数 EAX。

查看 off_100019040，能看到字符串[This is RDO]pics.practicalmalwareanalysis.com。

```
.data:10019040 off_10019040 dd offset aThisIsRdoPicsP
; DATA XREF: sub_10001656:loc_10001722↑r
; sub_10001656:FB↑r ...
; "[This is RDO]pics.practicalmalwareanalysis"...

.data:10019194 aThisIsRdoPicsP db '[This is RDO]pics.practicalmalwareanalysis.com',0
; DATA XREF: .data:off_10019040↑o
```

对字符串的指针又加上了 0xD 字节的偏移，因此 gethostbyname 使用的 EAX 是一个指向域名 pics.practicalmalwareanalysis.com 的指针。

这个调用会对该域名发起一个 DNS 请求，以获得其 IP 地址。

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

按 G 跳转到 0x10001656。看到绿色的部分是识别出来的局部变量。一共是 23 个，不包含最后一行的 lpThreadParameter，因为它是传入的参数。

```
.text:10001656 ; DWORD __stdcall sub_10001656(LPVOID lpThreadParameter)
.text:10001656 sub_10001656 proc near ; DATA XREF: DllMain(x,x,x)+C8↓o
.text:10001656
.text:10001656 var_675 = byte ptr -675h
.text:10001656 var_674 = dword ptr -674h
.text:10001656 hModule = dword ptr -670h
.text:10001656 timeout = timeval ptr -66Ch
.text:10001656 name = sockaddr ptr -664h
.text:10001656 var_654 = word ptr -654h
.text:10001656 in = in_addr ptr -650h
.text:10001656 Str1 = byte ptr -644h
.text:10001656 var_640 = byte ptr -640h
.text:10001656 CommandLine = byte ptr -63Fh
.text:10001656 Str = byte ptr -63Dh
.text:10001656 var_638 = byte ptr -638h
.text:10001656 var_637 = byte ptr -637h
.text:10001656 var_544 = byte ptr -544h
.text:10001656 var_50C = dword ptr -50Ch
.text:10001656 var_500 = byte ptr -500h
.text:10001656 Buf2 = byte ptr -4FCh
.text:10001656 readfds = fd_set ptr -4BCh
.text:10001656 buf = byte ptr -3B8h
.text:10001656 var_3B0 = dword ptr -3B0h
.text:10001656 var_1A4 = dword ptr -1A4h
.text:10001656 var_194 = dword ptr -194h
.text:10001656 WSADATA = WSADATA ptr -190h
.text:10001656 lpThreadParameter = dword ptr 4
.text:10001656
```

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

根据参数引用为正的偏移值，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？

通过 Strings 窗口，搜索 cmd.exe /c，可以观察到该字符串位于 xdoors_d 节中的 0x10095B34 处。

Address	Length	Type	String
xdoors_d:10095B34	0000000D	C	cmd.exe /c

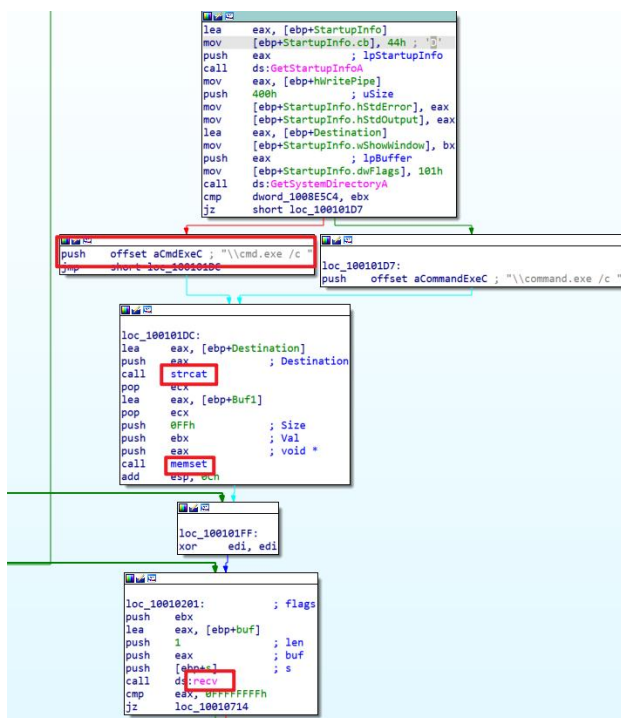
```
xdoors_d:10095B31 align 4
xdoors_d:10095B34 aCmdExeC db '\cmd.exe /c ',0 ; DATA XREF: sub_1000FF58+278↑o
xdoors_d:10095B41 align 4
xdoors_d:10095B44 ; char aHiMasterDDDDDD[]
xdoors_d:10095B44 aHiMasterDDDDDD db 'Hi,Master [%d/%d/%d %d:%d:%d]',0Dh,0Ah
xdoors_d:10095B44 ; DATA XREF: sub_1000FF58+145↑o
```

8. 在引用cmd.exe /c 的代码所在的区域发生了什么？

查看交叉引用，有且仅有 0x100101D0 一处，此时，字符串被压到栈上：

```
.text:100101C8      cmp     dword_1008E5C4, eax
.text:100101CF      jz      short loc_100101D7
.text:100101D0      push    offset aCmdExeC ; "\\cmd.exe /c "
.text:100101D5      jmp     short loc_100101DC
.text:100101D7
```

从这个函数的图形模式，可以看到一系列的 memcmp 函数被用于比较如 cd、exit、install、inject 和 uptime 等的字符串。查看一下这个函数发起的调用，可以看到有一系列的 recv 和 send 调用。



还可以看到在该字符串被引用之前的 0x1001009D 处，有一个字符串 This Remote Shell Session。

```
.text:10010097      lea     eax, [ebp+Str]
.text:1001009D      push    offset aHiMasterDDDDDD ; "Hi,Master [%d/%d/%d %d:%d:%d]\r\nWelcome..."
.text:100100A2      push    eax ; Buffer

xdoors_d:10095B44 ; char aHiMasterDDDDDD[]
xdoors_d:10095B44 aHiMasterDDDDDD db 'Hi,Master [%d/%d/%d %d:%d:%d]',0Dh,0Ah
xdoors_d:10095B44 ; DATA XREF: sub_1000FF58+145f0
xdoors_d:10095B44 db 'Welcome Back...Are You Enjoying Today?',0Dh,0Ah
xdoors_d:10095B44 db 0Dh,0Ah
xdoors_d:10095B44 db 'Machine UpTime [%-.2d Days %-.2d Hours %-.2d Minutes %-.2d Secon'
xdoors_d:10095B44 db 'ds]',0Dh,0Ah
xdoors_d:10095B44 db 'Machine IdleTime [%-.2d Days %-.2d Hours %-.2d Minutes %-.2d Seco'
xdoors_d:10095B44 db 'nds]',0Dh,0Ah
xdoors_d:10095B44 db 0Dh,0Ah
xdoors_d:10095B44 db 'Encrypt Magic Number For This Remote Shell Session [0x%02x]',0Dh,0Ah
xdoors_d:10095B44 db 0Dh,0Ah,0
xdoors_d:10095C5C ; char asc_10095C5C[]
```

因此，可以猜测正在查看的是一个远程 shell 会话函数。

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？




dword_1008E5C4 是一个全局变量，在 0x100101C8 处双击它，来到内存中的 0x1008E5C 处，这位于 DLL 文件的.data 节。


```

.text:100101C2      call     ds:GetSystemDirectoryA
.text:100101C8      cmp     dword_1008E5C4, ebx
.text:100101CF      iz      short loc_100101D7
.data:1008E5C4 dword_1008E5C4 dd ? ; DATA XREF: sub_10001656+22↑w
.data:1008E5C4      ; sub_10007312+E↑r ...

```

检查交叉引用，可以发现它被引用了 3 次，但只有一次修改它的值。

xrefs to dword_1008E5C4				
Direct	Typ	Address	Text	
	Up w	sub_10001656+22	mov	dword_1008E5C4, eax
	Up r	sub_10007312+E	cmp	dword_1008E5C4, edi
	Up r	sub_1000FF58+270	cmp	dword_1008E5C4, ebx

修改其值的代码如下。EAX 被赋给 dword_1008E5C4，而 EAX 是前一条指令函数调用的返回值。

```

.text:10001673      call     sub_10003695
.text:10001678      mov     dword_1008E5C4, eax

```

双击 sub_10003695 在反汇编窗口中查看该函数。该函数包括了一个 GetVersionEx 的调用，后者用于获取当前操作系统版本的信息，如其中将 dwPlatformId 与数字 2 进行比较，来确定如何设置 AL 寄存器。如果 PlatformId 为 VER_PLATFORM_WIN32_NT, AL 会被置位。这里只是简单地判断当前操作系统是否 Windows 2000 或 更高版本，我们可以得出结论，该全局变量通常会被置为 1。

```

.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

```

因此，操作系统版本号被保存在了 dword_1008E5C4 中。

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

在 0x10010452 处，可以看到与 robotwork 的 memcmp:

```

.text:10010444      push     9                ; Size
.text:10010446      lea     eax, [ebp+Buf1]
.text:1001044C      push    offset aRobotwork ; "robotwork"
.text:10010451      push    eax                ; Buf1
.text:10010452      call    memcmp
.text:10010457      add     esp, 0Ch
.text:1001045A      test    eax, eax
.text:1001045C      jnz     short loc_10010468
.text:1001045E      push    [ebp+s]           ; s
.text:10010461      call    sub_100052A2
.text:10010466      jmp     short loc_100103F6

```

如果该字符串为 robotwork，则不会跳转 loc_10010468 处，而会调用 sub_100052A2。查看 sub_100052A2 的代码，其参数为 socket 类型。也就是上图中 0x1001045E 处 push 进去的 [ebp+s]。在 100052E7 处，aSoftWareMicros 的值为 SOFTWARE\Microsoft\Windows\CurrentVersion。最后调用 RegOpenKeyEx 函数，读取该注册表值。

```

.text:100052A2      push    ebp
.text:100052A3      mov     ebp, esp
.text:100052A5      sub     esp, 60Ch
.text:100052AB      and     [ebp+Buffer], 0
.text:100052B2      push    edi
.text:100052B3      mov     ecx, 0FFh
.text:100052B8      xor     eax, eax
.text:100052BA      lea     edi, [ebp+var_60B]
.text:100052C0      and     [ebp+Data], 0
.text:100052C7      rep stosd
.text:100052C9      stosw
.text:100052CB      stosb
.text:100052CC      push    7Fh
.text:100052CE      xor     eax, eax
.text:100052D0      pop     ecx
.text:100052D1      lea     edi, [ebp+var_20B]
.text:100052D7      rep stosd
.text:100052D9      stosw
.text:100052DB      stosb
.text:100052DC      lea     eax, [ebp+phkResult]
.text:100052DF      push    eax                ; phkResult
.text:100052E0      push    0F003Fh           ; samDesired
.text:100052E5      push    0                ; uOptions
.text:100052E7      push    offset aSoftwareMicros ; "SOFTWARE\Microsoft\Windows\CurrentVe"
.text:100052EC      push    8000002h          ; hKey
.text:100052F1      call    ds:RegOpenKeyExA
.text:100052F7      test    eax, eax
.text:100052F9      jz      short loc_10005309
.text:100052FB      push    [ebp+phkResult] ; hKey
.text:100052FE      call    ds:RegCloseKey
.text:10005304      jmp     loc_100053F6

xdoors_d:10093A50  aSoftwareMicros db 'SOFTWARE\Microsoft\Windows\CurrentVersion',0
xdoors_d:10093A50                                ; DATA XREF: sub_10003EBC+40↑o
xdoors_d:10093A50                                ; sub_10003EBC+D3↑o ...

```

因此，注册表项 HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\WorkTime 和 WorkTimes 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？

在导出表中，找到 PSLIST 函数：

Name	Address	Ordinal
InstallRT	000000001000D847	1
InstallSA	000000001000DEC1	2
InstallSB	000000001000E892	3
PSLIST	0000000010007025	4
ServiceMain	000000001000CF30	5
StartEXS	0000000010007ECB	6
UninstallRT	000000001000F405	7
UninstallSA	000000001000EA05	8
UninstallSB	000000001000F138	9
DllEntryPoint	000000001001516D	[main entry]

查看代码：

```

. text:10007025 mov     dword_1008E5BC, 1
. text:1000702F call    sub_100036C3
. text:10007034 test    eax, eax
. text:10007036 jz      short loc_1000705B
. text:10007038 push    [esp+Str] ; Str
. text:1000703C call    strlen
. text:10007041 test    eax, eax
. text:10007043 pop     ecx
. text:10007044 jnz     short loc_1000704E
. text:10007046 push    eax
. text:10007047 call    sub_10006518
. text:1000704C jmp     short loc_1000705A
. text:1000704E ; -----
. text:1000704E loc_1000704E: push    [esp+Str] ; CODE XREF: PSLIST+1F1j
. text:10007052 push    0
. text:10007054 call    sub_1000664C
. text:10007059 pop     ecx
. text:1000705A loc_1000705A: pop     ecx ; CODE XREF: PSLIST+271j
. text:1000705B loc_1000705B: and     dword_1008E5BC, 0 ; CODE XREF: PSLIST+111j
. text:10007062 retn    10h
. text:10007062 PSLIST endp

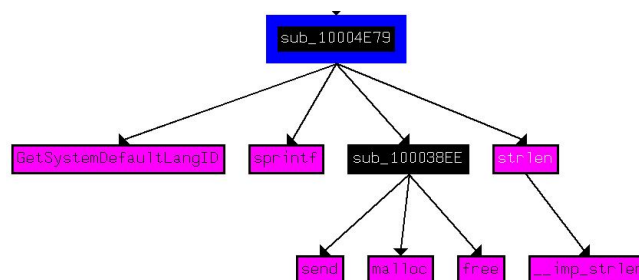
```

这个函数选择两条路径之一执行，这个选择取决于 sub_100036C3 的结果。sub_100036C3 函数检查操作系统的版本是 Windows Vista/7,或是 Windows XP/2003/2000。这两条代码路径都使用 CreateToolhelp32Snapshot 函数，从相关字符串和 API 调用来看，用于获得一个进程列表。这两条代码路径都通过 send 将进程列表通过 socket 发送。

因此，PSLIST 导出项可以通过网络发送进程列表，或者寻找该列表中某个指定的进程名并获取其信息。

12. 使用图模式来绘制出对 sub_10004E79 的交叉引用图。当进入这个函数时，哪个 API 函数可能被调用?仅仅基于这些 API 函数，你会如何重命名这个函数?

对 sub_10004E79 交叉引用的图示化结果如图所示：

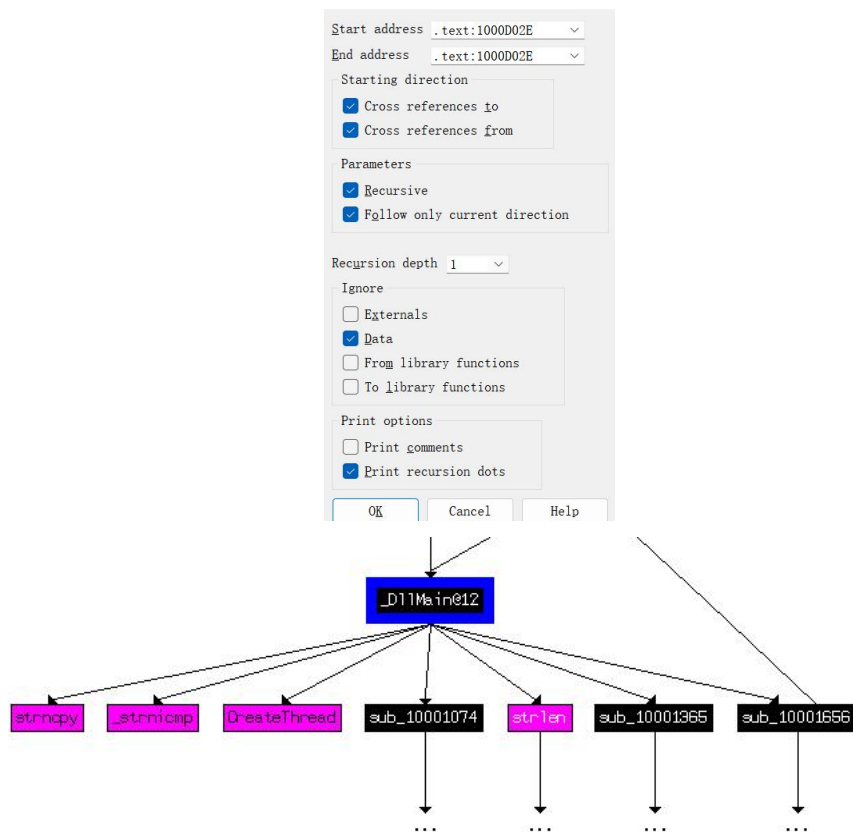


该函数调用了 GetSystemDefaultLangID 和 send。这一信息说明该函数可能通过 socket 发送语言标志，可以重命名为 send_languageID。

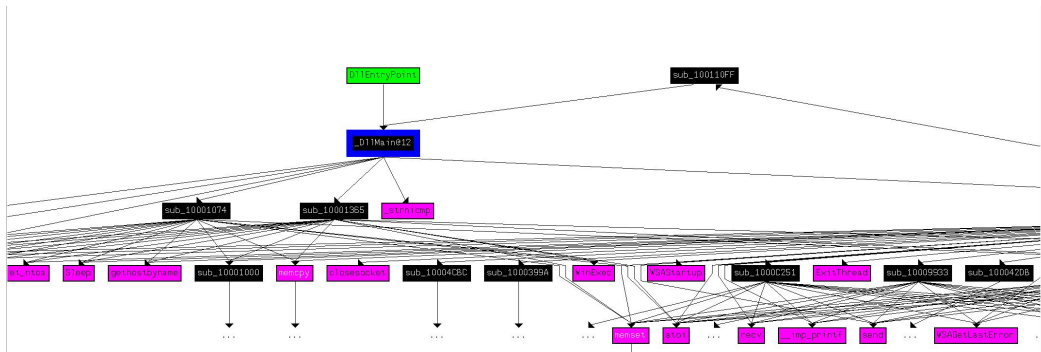
13. DllMain 直接调用了多少个 Windows API?多少个在深度为 2 时被调用?

要确定 Dllmain 调用多少 Windows API 函数，可以在 Xrefs From 中查看。

选取 Recursion depth(递归深度) 为 1，以显示其直接调用的函数：



如果将递归深度设置为 2，可以发现调用信息庞杂：



因此，D11Main 直接调用了 strcpy、strncpy、CreateThread 和 strlen 这些 API。进一步地，调用了非常多的 API，包括 Sleep、WinExec、gethostbyname,以及许多其他网络函数调用。

14. 在 0x10001358 处，有一个对 Sleep(一个使用一个包含要睡眠的毫秒数的参数的 API 函数)的调用。顺着代码向后看，如果这段代码执行，这个程序会睡眠多久？

查看相应代码。Sleep 只使用一个参数，也就是要休眠的毫秒数。该参数作为 EAX 被压入栈上。

```

.text:10001341 loc_10001341:                                     ; CODE XREF: sub_10001074+10F1j
.text:10001341                                     ; sub_10001074+1B01j ...
.text:10001341 mov     eax, off_10019020 ; "[This is CTI]30"
.text:10001346 add     eax, 0Dh
.text:10001349 push    eax ; String
.text:1000134A call    ds:atoi
.text:10001350 imul    eax, 3E8h
.text:10001356 pop     ecx
.text:10001357 push    eax ; dwMilliseconds
.text:10001358 call    ds:Sleep
.text:1000135E xor     ebp, ebp
.text:10001360 jmp     loc_100010B4
.text:10001360 sub_10001074 endp
.text:10001360

```

首先，off_10019020 被赋给 EAX，然后 EAX 自身加上 0xD。查看 off_10019020，它指向了一个字符串[This is CTI]30；加上偏移后，EAX 指向 30。接下来，调用 atoi，将字符串 30 转为数字 30。

接下来，EAX 乘 3E8h，也就是乘上 10 进制的 1000，得到 30000。所以，程序休眠的时间是 30000 毫秒（30 秒）。

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

查看调用代码，可以发现，三个参数分别为 6、1、2，代表含义分别为 protocol、type、af。

```

.text:100016FB loc_100016FB:                                     ; CODE XREF: sub_10001656+3741j
.text:100016FB                                     ; sub_10001656+A091j
.text:100016FB push    6 ; protocol
.text:100016FD push    1 ; type
.text:100016FF push    2 ; af
.text:10001701 call    ds:socket
.text:10001707 mov     edi, eax
.text:10001709 cmp     edi, 0FFFFFFFh
.text:1000170C jnz     short loc_10001722
.text:1000170E call    ds:WSAGetLastError
.text:10001714 push    eax
.text:10001715 push    offset aSocketGetlaste ; "socket() GetLastError reports %d\
.text:1000171A call    ds:__imp_printf
.text:10001720 pop     ecx
.text:10001721 pop     ecx

```

16. 使用 MSDN 页面的 socket 和 IDA Pro 中的命名符号常量，你能使参数更加有意义吗？在你应用了修改以后，参数是什么？

右键单击每个数，选择 UseSymbolic Constant，会弹出一个对话框，会列举出 IDA Pro 为这个特定值找到所有的对应常量。

在这里值 2 指的是 AF_INET，用于设置一个 IPv4 socket；值 1 指的是 SOCK_STREAM；值 6 指的是 IPPROTO_TCP。因此，这个 socket 会被配置为基于 IPv4 的 TCP 连接(常被用于 HTTP)

Symbol name	Value	Type library
IPPROTO_TCP	00000006	MS SDK (Windows 7)
SOCK_STREAM	00000001	MS SDK (Windows 7)
AF_INET	00000002	MS SDK (Windows 7)

将进行修改，如图所示：

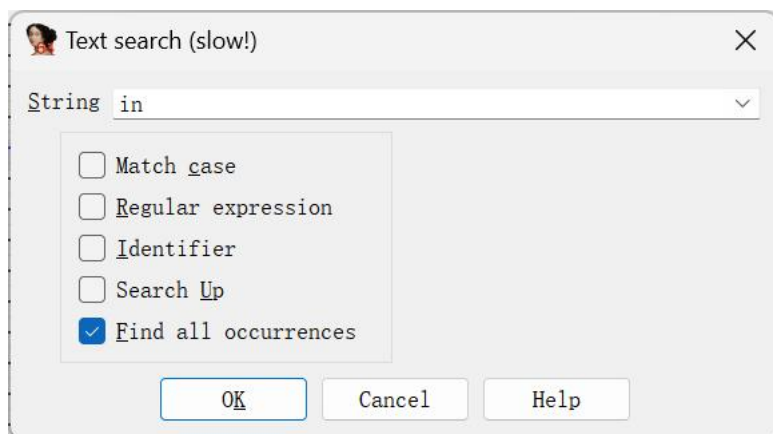
```

.text:100016FB      push     IPPROTO_TCP      ; protocol
.text:100016FD      push     SOCK_STREAM      ; type
.text:100016FF      push     AF_INET          ; af
.text:10001701      call     ds:socket

```

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?

搜索 `in` 指令:



可以发现, 在 `0x100061DB` 处有 `in` 指令:

```

.text:10006196      sub_10006196      sub_10006196      proc near      ; CODE XREF: InstallRT+20↓p
.text:100061DB      sub_10006196      in     eax, dx
.text:1000620C      sub_1000620C      ; int __cdecl sub_1000620C(char *Format, char ArgList)
.text:1000620C      sub_1000620C      call     ds:format

.text:100061C7      mov     eax, 564D5868h
.text:100061CC      mov     ebx, 0
.text:100061D1      mov     ecx, 0Ah
.text:100061D6      mov     edx, 5658h
.text:100061DB      in     eax, dx
.text:100061DC      cmp     ebx, 564D5868h

```

在 `0x100061C7` 处的 `mov` 指令将 `0x564D5868` 赋给 `EAX`。右击这个值, 可以看到它相当于 ASCII 字符串 `VMXh`, 这也就确认了这段代码是恶意代码采用的反虚拟机技巧。

18. 将你的光标跳转到 `0x1001D988` 处, 你发现了什么?

可以发现, 一些随机数据字节, 没有可读性和实际意义。

```

.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

```

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?

运行脚本后，0x1001D988 处的数据变为一些可读的内容。

.data:1001D987	dd	7
.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	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	20h

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？

在地址 0x1001D988 处，右键选择转换成 ASCII 字符串（或者按下 A 键），得到字符串 'xdoor is this backdoor, string decoded for Practical Malware Analysis Lab :)1234':

.data:1001D987	dd	7
.data:1001D988	db	'xdoor is t=<&u746>1::',27h,'yu! ',27h,'<;2u106:101u3:',27h,'u'
.data:1001D989	db	5

21. 使用一个文本编辑器打开这个脚本。它是如何工作的？

打开脚本：

```
sea = ScreenEA()

for i in range(0x00,0x50):
    b = Byte(sea+i)
    decoded_byte = b ^ 0x55
    PatchByte(sea+i,decoded_byte)
```

脚本的工作是：对长度为 0x50 字节的数据，用 0x55 分别与其进行异或，然后用 PatchByte 函数在 IDA Pro 中修改这些字节。

22. 编写 Yara 规则

根据以上分析和查看到的字符串信息，编写 Yara 规则如下：

```
rule Lab05_01 {
    meta:
        description = "Lab05-01.dll"
```



```

strings:
    $s1 = "\\cmd.exe /c " fullword ascii
    $s2 = "SOFTWARE\\Microsoft\\Windows\\CurrentVersion" fullword ascii
    $s3 = "Uninject '%s' From Process '%s' " fullword ascii
    $s4 = "error on get process info. " fullword ascii
    $s5 = "Process '%s' Not Found ,Inject Failed" fullword ascii
    $s6 = "Inject '%s' To Process '%s' Failed" fullword ascii
    $s8 = "Inject '%s' To' %x' Process '%s'" fullword ascii
    $s9 = "\\command.exe /c " fullword ascii

condition:
    uint16(0) == 0x5a4d and
    uint32(uint32(0x3c))==0x00004550 and filesize < 400KB and
    6 of them
}

```

运行 Yara 规则，能够扫描到对应的恶意代码：

```

PS D:\NKU\23Fall\恶意代码分析与防治技术\yara-4.3.2-2150-win64> ./yara64 lab05.yar virus
Lab05_01 virus\Lab05-01.dll

```

23. 尝试编写 IDA Python 脚本辅助样本分析。

(1) 辅助查找函数的交叉引用

编写脚本如下：

```

import idaapi
import idautils

def find_and_display_xrefs(target_function_name):
    target_function_ea = idaapi.get_name_ea(0, target_function_name)

    if target_function_ea != idaapi.BADADDR:
        xrefs = list(idautils.XrefsTo(target_function_ea))

        if xrefs:
            print(f'找到 {len(xrefs)} 个对 {target_function_name} 的交叉引用: ")
            for xref in xrefs:
                print(f'来自 {idaapi.get_func_name(xref.frm)}, 地址: 0x{target_function_ea:X}')
        else:
            print(f'未找到对 {target_function_name} 的交叉引用。")

```

```

else:
    print(f'未找到函数 {target_function_name}。')

if __name__ == "__main__":
    target_function_name = "gethostbyname"

    find_and_display_xrefs(target_function_name)

```

运行脚本结果如下：

```

找到 18 个对 gethostbyname 的交叉引用:
来自 sub_10001074, 地址: 0x100163CC
来自 sub_10001074, 地址: 0x100163CC
来自 sub_10001074, 地址: 0x100163CC
来自 sub_10001365, 地址: 0x100163CC
来自 sub_10001365, 地址: 0x100163CC
来自 sub_10001365, 地址: 0x100163CC
来自 sub_10001656, 地址: 0x100163CC
来自 sub_1000208F, 地址: 0x100163CC
来自 sub_10002CCE, 地址: 0x100163CC
来自 sub_10001074, 地址: 0x100163CC
来自 sub_10001074, 地址: 0x100163CC
来自 sub_10001074, 地址: 0x100163CC
来自 sub_10001365, 地址: 0x100163CC
来自 sub_10001365, 地址: 0x100163CC
来自 sub_10001365, 地址: 0x100163CC
来自 sub_10001656, 地址: 0x100163CC
来自 sub_1000208F, 地址: 0x100163CC
来自 sub_10002CCE, 地址: 0x100163CC

```

Python

(2) 打印函数地址和函数指令：

```

import idaapi
import idutils

target_function_name = "sub_10004E79"

target_function_ea = idc.get_name_ea_simple(target_function_name)

if target_function_ea != idc.BADADDR:
    print(f'函数 {target_function_name} 的地址: {target_function_ea:X}')

    for ea in idutils.FuncItems(target_function_ea):
        disasm = idc.GetDisasm(ea)
        print(f'{ea:X}: {disasm}')

else:

```

```
print(f"没有找到函数 {target_function_name}")
```

脚本运行情况:

```
-----
函数 sub_10004E79 的地址: 10004E79
10004E79: push    ebp
10004E7A: mov     ebp, esp
10004E7C: sub     esp, 400h
10004E82: and     [ebp+Buffer], 0
10004E89: push    edi
10004E8A: mov     ecx, 0FFh
10004E8F: xor     eax, eax
10004E91: lea     edi, [ebp+var_3FF]
10004E97: rep stosd
10004E99: stosw
10004E9B: stosb
10004E9C: call    ds:GetSystemDefaultLangID
10004EA2: movzx   eax, ax
10004EA5: push    eax
10004EA6: lea     eax, [ebp+Buffer]
10004EAC: push    offset aLanguageId0xX; "\r\n\r\n[Language:] id:0x%x\r\n\r\n"
10004EB1: push    eax; Buffer
10004EB2: call    ds:sprintf
10004EB8: add     esp, 0Ch
10004EBB: lea     eax, [ebp+Buffer]
10004EC1: push    0
10004EC3: push    eax; Str
10004EC4: call    strlen
10004EC9: pop     ecx
10004ECA: push    eax; len
10004ECB: lea     eax, [ebp+Buffer]
10004ED1: push    eax; int
10004ED2: push    [ebp+s]; s
10004ED5: call    sub_100038EE
10004EDA: add     esp, 10h
10004EDD: pop     edi
10004EDE: leave
10004EDF: retn
```

四、实验结论及心得体会

本次实验使用 IDA PRO 分析病毒样本，并使用 IDA Python 辅助分析。