南開大學

恶意代码分析与防治课程实验报告

实验七：ollydbg

学　　院　网络空间安全学院

专　　业　信息安全

学　　号　2111033

姓　　名　艾明旭

班　　级　信息安全一班

一、实验目的

实验目的：通过使用 OllyDbg 进行恶意代码分析与实战，达到以下目标：

理解恶意代码的工作原理：通过分析恶意代码执行过程，了解其背后的技术和攻击手段。掌握 OllyDbg 的使用：学习如何使用 OllyDbg 这款强大的调试器工具，用于动态分析和调试恶意代码。分析恶意代码的行为：通过 OllyDbg 的调试功能，跟踪恶意代码的执行流程，探查其具体的行为和影响。发现漏洞和弱点：通过分析恶意代码的漏洞和弱点，帮助提高对系统和应用程序的安全性，及时修补潜在的威胁。

二、实验原理

1、 实验环境

Windows xp，VMWARE，Windows11 ,win8.1

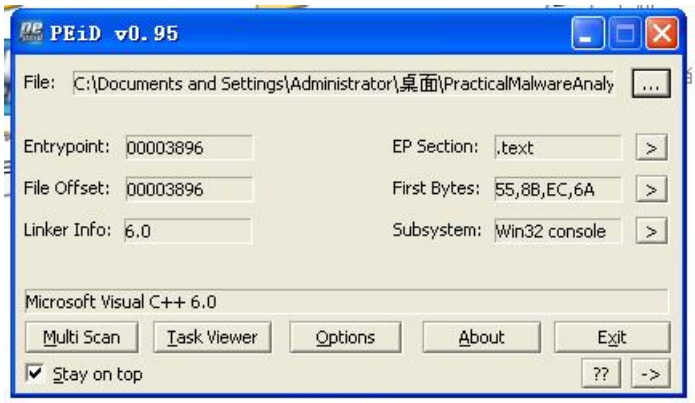
2、 实验工具

Ida pro 6.6 ida python,yara

三、实验过程

Lab9-1 分析

1、首先查壳；



无壳；

2、拖进 IDA Pro 查看导入函数；

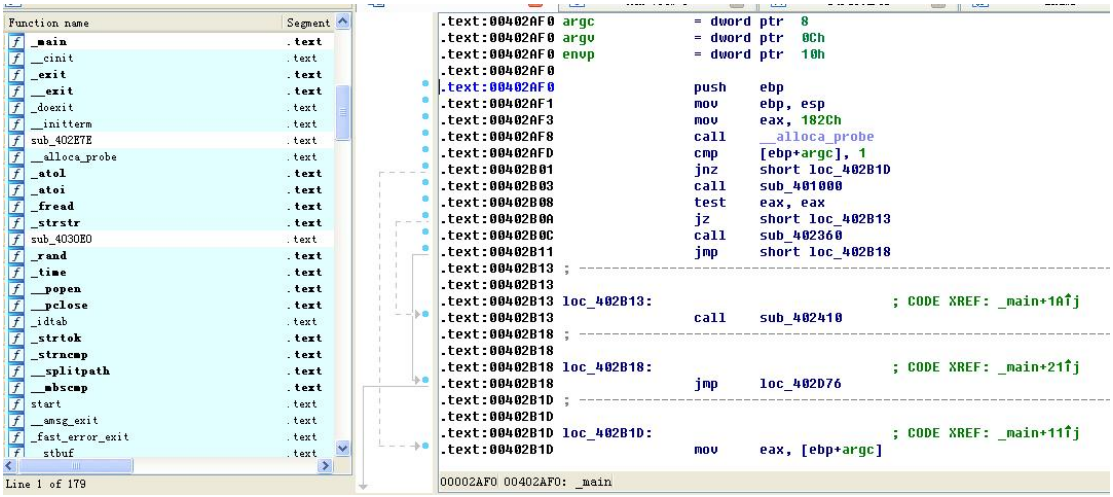
太多，就不截图了。但是我们可以看见很多特性函数，比如：

OpenSCManager 和 OpenServiceA 等服务函数；

Reg 开头的注册表函数；

Get 和 Set 开头获取权限或者信息的函数；

最后是一些 WSAStartup、connect、socket、send 等网络传输函数；



综上判断这个程序可能出现的操作涉及注册表以及注册服务工作，目的是进行网络传输一些数据；

3、shift+f12 快捷键查看字符串；

command.com 和 cmd.exe 这种敏感指令；

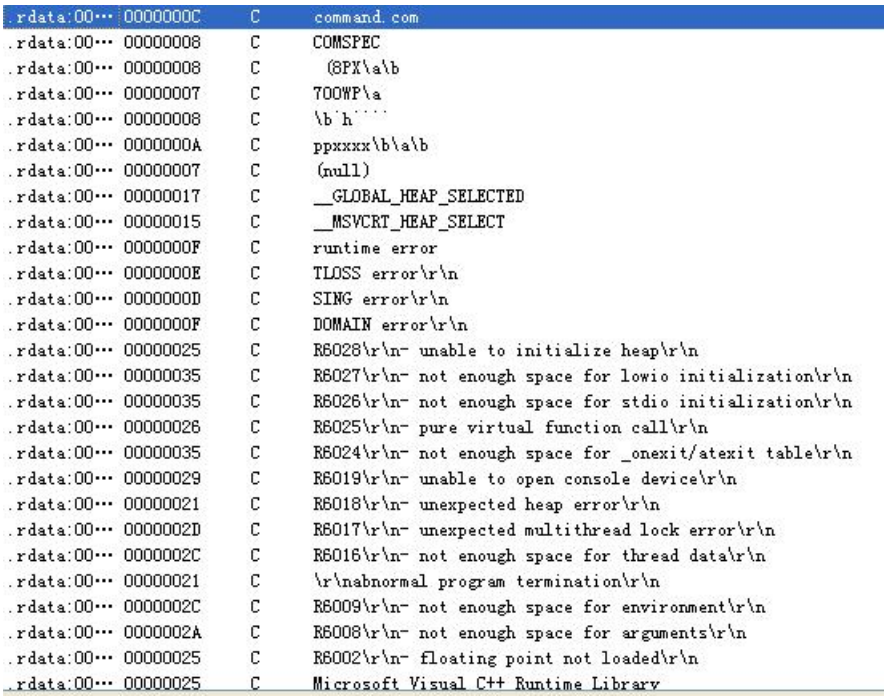
.com、.bat 和.cmd 这种有趣的文件后缀；

http/1.0\r\n\r\n 和 GET 这种网络关键字；

一个网址 http://www.practicalmalwareanalysis.com；

%SYSTEMOOT%\\SYSTEM32\\敏感路径；

更加确认了有网络行为，并且有远程指令的嫌疑；



4、分析行为；

使用 IDA Pro 查看 main()入口，再使用 OllyDbg 跟踪；

```
00402AF0 ; int __cdecl main(int argc, const char **argv, const char **envp)
00402AF0 _main           proc near               ; CODE XREF: start+AF↓p
00402AF0
00402AF0 var_182C        = dword ptr -182Ch
00402AF0 var_1828        = dword ptr -1828h
00402AF0 var_1824        = dword ptr -1824h
00402AF0 var_1820        = dword ptr -1820h
00402AF0 var_181C        = byte ptr -181Ch
00402AF0 var_141C        = byte ptr -141Ch
00402AF0 var_101C        = byte ptr -101Ch
00402AF0 var_C1C         = byte ptr -0C1Ch
00402AF0 var_81C         = dword ptr -81Ch
00402AF0 var_818         = dword ptr -818h
00402AF0 var_814         = dword ptr -814h
00402AF0 var_810         = dword ptr -810h
00402AF0 var_80C         = dword ptr -80Ch
00402AF0 var_808         = byte ptr -808h
00402AF0 lpServiceName   = dword ptr -408h
00402AF0 ServiceName     = byte ptr -404h
00402AF0 var_4           = dword ptr -4
00402AF0 argc            = dword ptr  8
00402AF0 argv            = dword ptr  0Ch
00402AF0 envp            = dword ptr  10h
```

问题 1.如何让这个恶意代码安装自身?

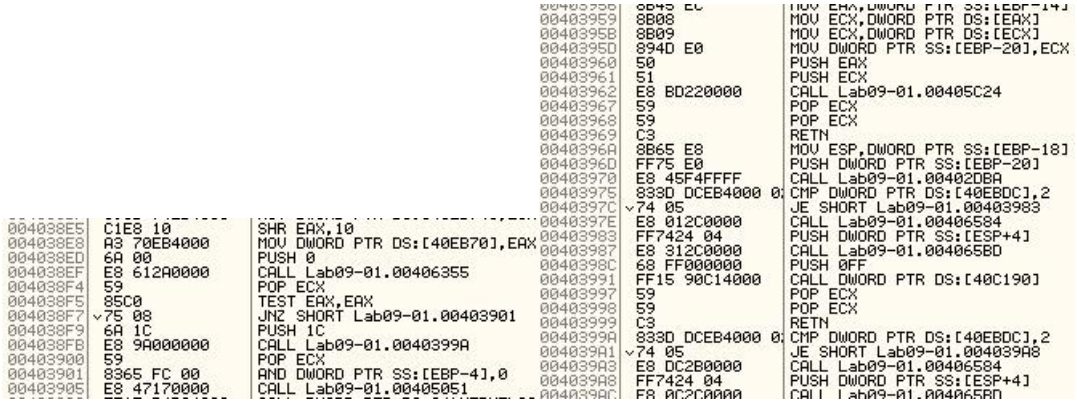
我们先打开 OllyDbg，我们可以看到我们停到了这里这个地方，这里对我们来说并不是什么重要的东西，我们可以按 F8(step-over)往后慢慢一步一步走，其实这时候按 F7（step-into)是最保险的，因为你不知道什么时候会调用到函数，然后如果你想回退到开始执行的地方，可以按 Ctrl+F2 来回溯到函数一开始执行的地方

```
00403893  5E              POP ESI
00403894  5B              POP EBX
00403895  C3              RETN
00403896  55              PUSH EBP
00403897  8BEC            MOV EBP,ESP
00403899  6A FF           PUSH -1
0040389B  68 88B14000     PUSH Lab09-01.0040B188
004038A0  68 AC644000     PUSH Lab09-01.004064AC
004038A5  64:A1 00000000  MOV EAX,DWORD PTR FS:[0]
004038AB  50              PUSH EAX
004038AC  64:8925 0000000 MOV DWORD PTR FS:[0],ESP
004038B3  83EC 10         SUB ESP,10
004038B6  53              PUSH EBX
004038B7  56              PUSH ESI
004038B8  57              PUSH EDI
004038B9  8965 E8         MOV DWORD PTR SS:[EBP-18],ESP
004038BC  FF15 B8B04000   CALL DWORD PTR DS:[<&KERNEL32.GetVersio KERNEL32.GetVersion
004038C2  33D2            XOR EDX,EDX
004038C4  8AD4            MOV DL,AH
004038C6  8915 7CEB4000   MOV DWORD PTR DS:[40EB7C],EDX
004038CC  8BC8            MOV ECX,EAX
004038CE  81E1 FF000000   AND ECX,0FF
004038D4  890D 78EB4000   MOV DWORD PTR DS:[40EB78],ECX
004038DA  C1E1 08         SHL ECX,8
004038DD  03CA            ADD ECX,EDX
004038DF  890D 74EB4000   MOV DWORD PTR DS:[40EB74],ECX
004038E5  C1E8 10         SHR EAX,10
004038E8  A3 70EB4000     MOV DWORD PTR DS:[40EB70],EAX
004038ED  6A 00           PUSH 0
004038EF  E8 612A0000     CALL Lab09-01.00406355
004038F4  59              POP ECX
```

然后我们边按边对照这 IDA 的汇编代码看，不然光看 OD 的代码会把人看傻的

我们先来到第一个 call 的地方。

这里 OD 是标识了调用的是 kernel32.GetVersion 的函数



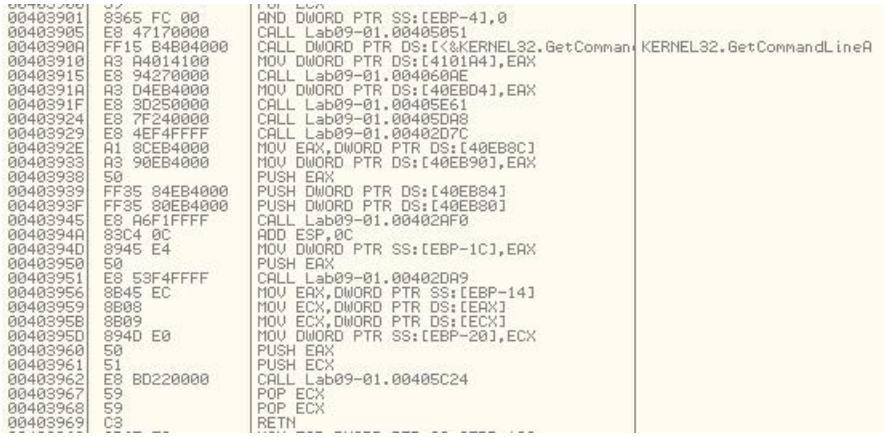
到现在这些函数都是程序执行前的初始化，这里是获取导入库的地址的

然后我们继续往下，忽略那些有的没的函数，来到第二个调用 CALL 这里，这里具体是什么

我们可以发现这里其实也是代码的初始化过程。

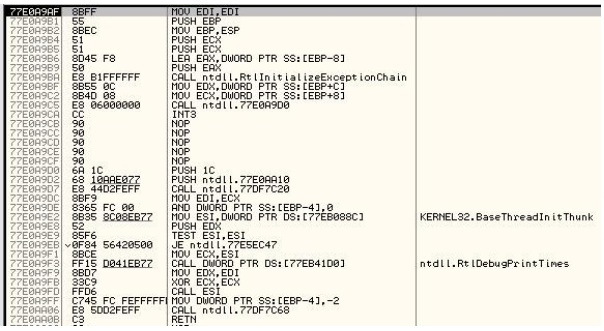
然后我们来到这个 CALL 的地方，这个也是初始化的东西，但是我们注意到下面这个 CALL 的函数是 GetCommandLineA，这个函数是获得用户输入的函数，说明这个程序需要用户提供一个参数。

然后我们继续往下会发现。



在 3915 这个地方，然后这里有个 CALL，我们按 F7 进去看看

进去发现这个也是初始化的过程。



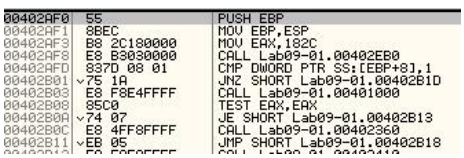
下面就是这三个函数的 CALL，我们进去第一个调用 405E61 处，然后我们会发现这个函数其实就是调用模块的。第二个调用也是用于获取文件名字的。第三个调用获得了当前进程的 ID，这其实也是程序执行过程中，初始化的过程。

继续往下分析，这里有三个 push，是 Lab09-01.00402AF0 调用的三个入参，右边的注释已经帮我们标出来了，后面我们分析会发现，其实这个调用 Lab09-01.00402AF0 就是 main 函数。然后我们就进入这个函数看看

这是进入的第一个 CALL，我们注意到这个 CALL 后面的一行的代码是 CMP DWORD PTR SS:[EBP+8], 1

这时候我们可以看看 IDA 了

```
:00402AF0 ; int __cdecl main(int argc, const char **argv, const char **envp)
:00402AF0 _main           proc near               ; CODE XREF: start+AF↓p
:00402AF0
:00402AF0 var_182C        = dword ptr -182Ch
:00402AF0 var_1828        = dword ptr -1828h
:00402AF0 var_1824        = dword ptr -1824h
:00402AF0 var_1820        = dword ptr -1820h
:00402AF0 var_181C        = byte ptr -181Ch
:00402AF0 var_141C        = byte ptr -141Ch
:00402AF0 var_101C        = byte ptr -101Ch
:00402AF0 var_C1C         = byte ptr -0C1Ch
:00402AF0 var_81C         = dword ptr -81Ch
:00402AF0 var_818         = dword ptr -818h
:00402AF0 var_814         = dword ptr -814h
:00402AF0 var_810         = dword ptr -810h
:00402AF0 var_80C         = dword ptr -80Ch
:00402AF0 var_808         = byte ptr -808h
:00402AF0 lpServiceName   = dword ptr -408h
:00402AF0 ServiceName     = byte ptr -404h
:00402AF0 var_4           = dword ptr -4
:00402AF0 argc            = dword ptr  8
:00402AF0 argv            = dword ptr  0Ch
:00402AF0 envp            = dword ptr  10h
:00402AF0
:00402AF0                 push    ebp
:00402AF1                 mov     ebp, esp
:00402AF3                 mov     eax, 182Ch
:00402AF8                 call    __alloca_probe
:00402AFD                 cmp     [ebp+argc], 1
:00402B01                 jnz     short loc_402B1D
:00402B03                 call    sub_401000
```



IDA 里面，标黄那个 CALL 下面的 CMP 是不是很眼熟

我们看上面的定义，argc=8，转换一下这个语句就是[ebp+8]，这就和 OD 对应上了

其实上面那个 OD 的第一个调用，就是__alloca_probe 这个函数，这个函数是分配栈空间用的，于是我们就大概知道这是在比较你的有没有输入参数，如有有参数的话，argc 就不会等于 1，如果没有就是 1。现在我们没有输入任何参数，然后[ebp+argc]其实=1，所以 cmp 相减之后，结果为 0，于是 ZF=1，则 jnz 不跳转

我们可以会到 OD 里面一步一步看看有没有跳转

的确，OD 里面并没有跳转，继续往下执行了。

这时候如果只看 OD 的数据，怎么知道 DWORD PTR SS:[EBP+8]就是 argc 呢，如果你想这种硬分析的话，我们可以回到进入 main 函数之前

这里在吊用 Lab09-01.00402AF0 之前 push 入栈了三个参数，这个函数就是我们标记为 main 的函数。然后我们会发现这个 main 函数的三个入参都被 OD 标注了类型，然后还要我们清楚的一点就是这个每个参数的大小问题，不清楚的可以看看 OD 的标注，这里我们可以在 Arg1 和 Arg2 这里看出来每个参数的大小是多少

Arg1 的地址是 DS:[40EB84]，然后 Arg2 的地址是 DS:[40EB80]，从这里我们可以看出这个代码中，每个参数是占 4 个字节的大小

然后这里有个小的 skill，在汇编函数调用中，我们压入这三个函数完之后，并不是马上就调用了这个函数，而是还要压入函数的返回地址

我们按照 param3，param2，param1 的次序依次在栈中压入参数，对应我们这里就是 Arg3，Arg2，Arg1 的次序。

```
0040392E  A1 8CEB4000     MOV EAX,DWORD PTR DS:[40EB8C]
00403933  A3 90EB4000     MOV DWORD PTR DS:[40EB90],EAX
00403938  50              PUSH EAX
00403939  FF35 84EB4000   PUSH DWORD PTR DS:[40EB84]
0040393F  FF35 80EB4000   PUSH DWORD PTR DS:[40EB80]
00403945  E8 A6F1FFFF     CALL Lab09-01.00402AF0
```

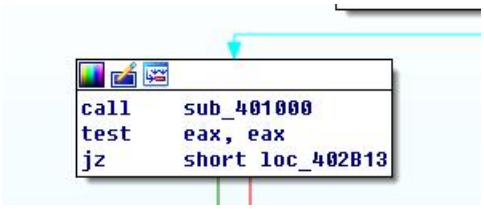
然后这时到了程序转移到被调用函数中执行，这个期间，地址 ebp+0 指向的是外面那个函数的 ebp 指针，ebp+4 指向的是 RET 返回地址，ebp+8 就是我们的第一个入参，也就是这里的 Arg1。

所以这里为啥 EBP+8 对应 argc 就是这个道理

然后因为我们现在调试的时候并没有输入任何的入参，所以这里我们的 argc 是等于 1 的

所以我们这里并不会跳转。

而是继续往下执行，然后从 00402B03 开始的话，就到了 IDA 的这里




```
00401001  8BEC            MOV EBP,ESP
00401003  83EC 08         SUB ESP,8
00401006  8D45 F8         LEA EAX,DWORD PTR SS:[EBP-8]
00401009  50              PUSH EAX
0040100A  68 3F000F00     PUSH 0F003F
0040100F  6A 00           PUSH 0
00401011  68 40C04000     PUSH Lab09-01.0040C040                ASCII "SOFTWARE\Microsoft
00401016  68 02000080     PUSH 80000002
0040101B  FF15 20B04000   CALL DWORD PTR DS:[<&ADVAPI32.RegOpenKe apphelp.74FDFE60
00401021  85C0            TEST EAX,EAX
00401023 v74 04           JE SHORT Lab09-01.00401029
00401025  33C0            XOR EAX,EAX
00401027 vEB 3D           JMP SHORT Lab09-01.00401066
00401029  6A 00           PUSH 0
0040102B  6A 00           PUSH 0
0040102D  6A 00           PUSH 0
0040102F  6A 00           PUSH 0
00401031  68 30C04000     PUSH Lab09-01.0040C030                ASCII "Configuration"
00401036  8B4D F8         MOV ECX,DWORD PTR SS:[EBP-8]
00401039  51              PUSH ECX
0040103A  FF15 24B04000   CALL DWORD PTR DS:[<&ADVAPI32.RegQueryV apphelp.74FE0A60
00401040  8945 FC         MOV DWORD PTR SS:[EBP-4],EAX
00401043  837D FC 00      CMP DWORD PTR SS:[EBP-4],0
00401047 v74 0E           JE SHORT Lab09-01.00401057
00401049  8B55 F8         MOV EDX,DWORD PTR SS:[EBP-8]
0040104C  52              PUSH EDX
0040104D  FF15 64B04000   CALL DWORD PTR DS:[<&KERNEL32.CloseHand KERNEL32.CloseHandle
00401053  33C0            XOR EAX,EAX
00401055 vEB 0F           JMP SHORT Lab09-01.00401066
00401057  8B45 F8         MOV EAX,DWORD PTR SS:[EBP-8]
0040105A  50              PUSH EAX
0040105B  FF15 64B04000   CALL DWORD PTR DS:[<&KERNEL32.CloseHand KERNEL32.CloseHandle
00401061  B8 01000000     MOV EAX,1
```

这里调用了 RegOpenKeyExA 这个函数，然后 OD 已经帮我们标注好了这个入参的个个值，这里比 IDA 人性得很多。这里是打开一个注册表的键，位置是 HKEY_LOCAL_MACHINE 这个地方，然后具体位置是在 SOFTWARE\Microsoft \XPS 这里

然后调用完之后就是测试这个调用是否成功了。这里 test 了一下返回值，一般返回值是成功就返回 0，假设调用成功了，and 之后，结果为 0，ZF=1，那么 je 跳转

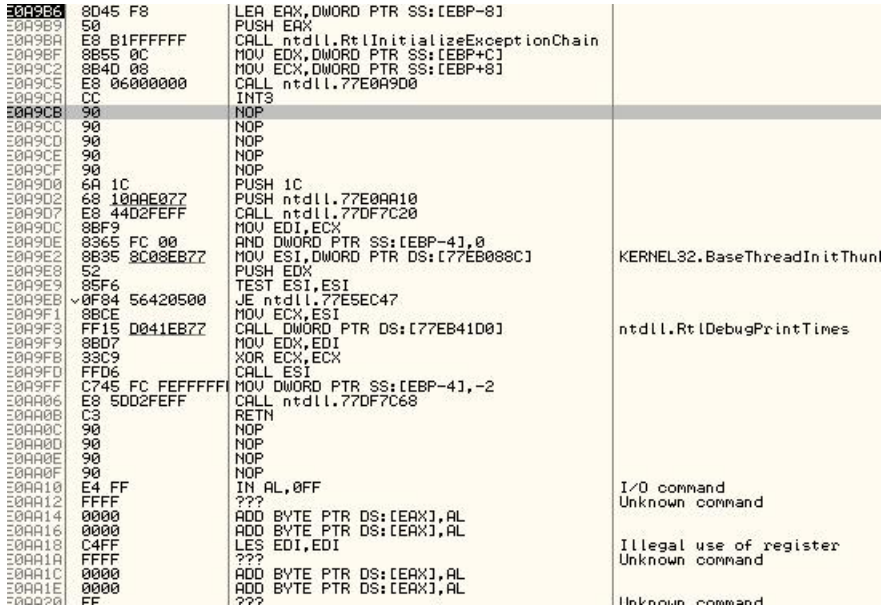
但是很不幸的是，我们 OD 调试的时候这个函数是返回失败的

```
Registers (FPU)                          <    <
EAX 00403896 Lab09-01.<ModuleEntryPoint>
ECX 00000000
EDX 00000000
EBX 7FFDE000
ESP 0018FFE8
EBP 0018FFEC
ESI 00000000
EDI 00000000

EIP 77E0A9B5 ntdll.77E0A9B5

C 0  ES 002B 32bit 0(FFFFFFFF)
P 0  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 0  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 7FFDD000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr ERROR_ENVVAR_NOT_FOUND (000000CB)

EFL 00000202 (NO,NB,NE,A,NS,PO,GE,G)

ST0 empty 0.0
ST1 empty 0.0
ST2 empty 0.0
ST3 empty 0.0
ST4 empty 0.0
ST5 empty 0.0
ST6 empty 0.0
ST7 empty 0.0
               3 2 1 0      E S P U O Z D I
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0 0  (G
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1
```

函数的返回值在 EAX 中存储，这时候是 2

然后就是函数把 eax 置为 0 之后返回了，于是我们大概知道这个函数的一半是干什么的了就是检测系统中是否存在那个注册表键，如果没有的话，就退出这个函数，并返回 0，那如果成功呢，我们这里手动修改了一下 eax 看看

我们将这个 ZF 标志位手动置 1，然后执行，我们到了这个地方。



这里调用了一个注册表查询的函数，将名为 Configuration 的值查出来

这里调用完成之后我们手动将返回值也置为 0，因为这时候这个键都不存在，哪里会查询成功。然后就是一些返回值比较函数的作用，注意这里的 ebp-4 的位置，这里一般是属于临时空间。这里的 JE 其实和 JZ 是一样的，如果相等就跳转

如果调用失败之后，也会返回 0。如果两个函数都调用成功，就返回

```
lea     eax, [ebp+var_C1C]
push    eax
lea     ecx, [ebp+var_181C]
push    ecx
lea     edx, [ebp+var_101C]
push    edx
lea     eax, [ebp+var_141C]
push    eax
push    offset aKSHSPSPerS ; "k:%s h:%s p:%s per:%
call    sub_402E7E
add     esp, 14h
```

然后这里我们为了节约分析时间，分析过后，其实这里的 sub_401000 如果调用失败，jz 跳转之后，就会去执行一个 Lab09-01.00402410 的函数。

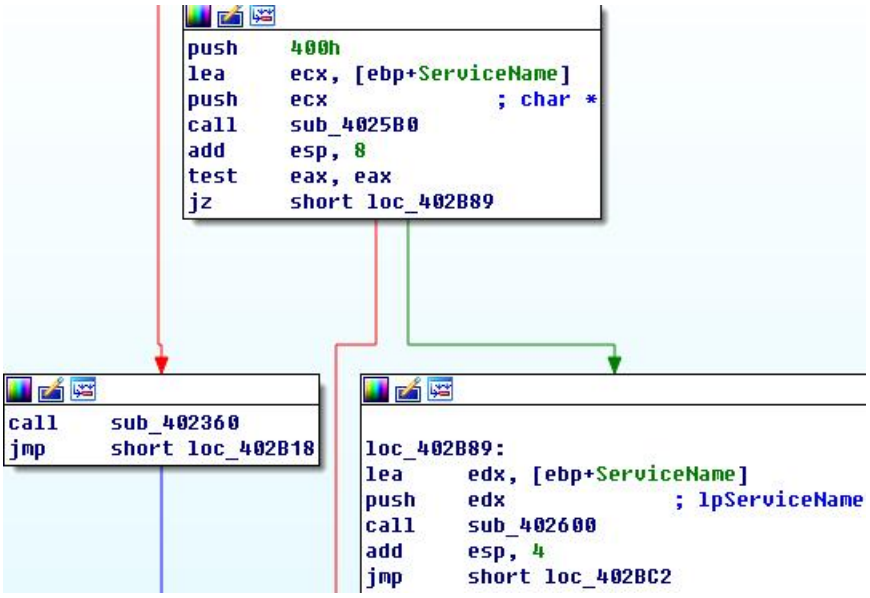
其实也就是 IDA 中的 sub_402410。这里用 ShellExecuteA 打开了 cmd.exe，这应该是这个恶意程序的一种伪造，如果用户没有输入任何的参数，或者没有发现那个注册表项，再或者查询注册表失败，就来执行这个 ShellExecuteA 通过 cmd.exe 来运行一些命令。

关键现在我们要找出这个命令是什么，然后我们现在回到这个函数的开始，开始我们的分析。

这里是开头的地方，一样的先是做了一些栈的初始化，将 ebp 压栈备份，然后把 ebp 指向 esp。然后做完这些后把 esp 相减 208，将 esp 往低地址的地方偏移了 208 个地址，4 个地址存储一个字节，有 52 个字节的存储空间

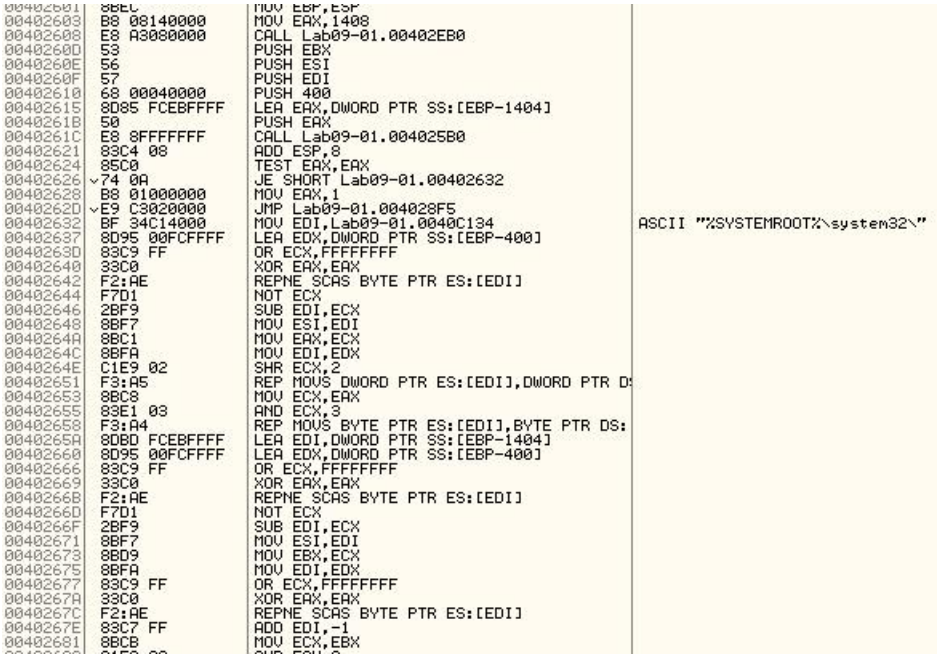
然后我们从函数中返回之后就是来到了这里，这有个 test，我们成功了，所以返回值是 0，然后 test 之后 ZF=0，JE 跳转

如果这里不是返回 0，那么整个这个函数会跳转之后结束并返回-1，跳转之后来到这里

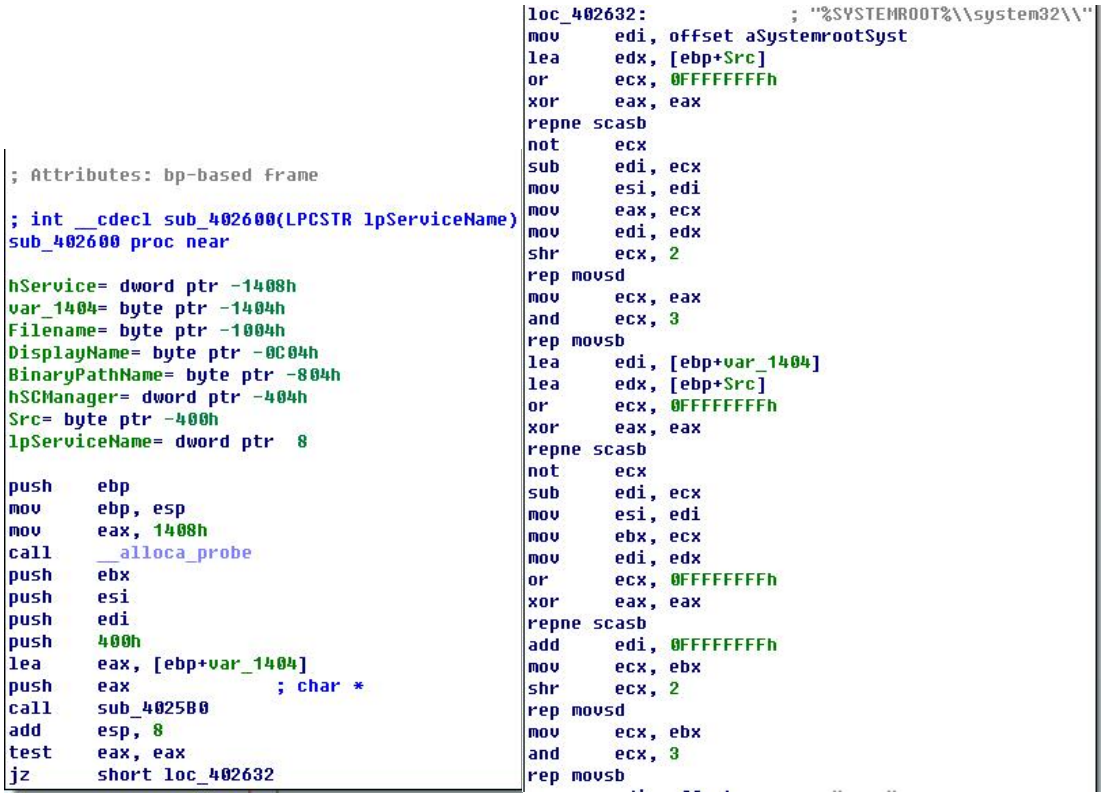


我们可以大概看出来，这个 EDX 其实就是刚刚上面那个函数处理后的返回结果 Lab09-01

函数进来之后的是这样的（这个函数很大）



对应 IDA 中就是这样的



```
; Attributes: bp-based frame

; int __cdecl sub_402600(LPCSTR lpServiceName)
sub_402600 proc near

hService= dword ptr -1408h
var_1404= byte ptr -1404h
Filename= byte ptr -1004h
DisplayName= byte ptr -0C04h
BinaryPathName= byte ptr -804h
hSCManager= dword ptr -404h
Src= byte ptr -400h
lpServiceName= dword ptr  8

push    ebp
mov     ebp, esp
mov     eax, 1408h
call    __alloca_probe
push    ebx
push    esi
push    edi
push    400h
lea     eax, [ebp+var_1404]
push    eax             ; char *
call    sub_4025B0
add     esp, 8
test    eax, eax
jz      short loc_402632
```

```
loc_402632:             ; "%SYSTEMROOT%\\system32\\"
mov     edi, offset aSystemrootSyst
lea     edx, [ebp+Src]
or      ecx, 0FFFFFFFFh
xor     eax, eax
repne scasb
not     ecx
sub     edi, ecx
mov     esi, edi
mov     eax, ecx
mov     edi, edx
shr     ecx, 2
rep movsd
mov     ecx, eax
and     ecx, 3
rep movsb
lea     edi, [ebp+var_1404]
lea     edx, [ebp+Src]
or      ecx, 0FFFFFFFFh
xor     eax, eax
repne scasb
not     ecx
sub     edi, ecx
mov     esi, edi
mov     ebx, ecx
mov     edi, edx
or      ecx, 0FFFFFFFFh
xor     eax, eax
repne scasb
add     edi, 0FFFFFFFFh
mov     ecx, ebx
shr     ecx, 2
rep movsd
mov     ecx, ebx
and     ecx, 3
rep movsb
```

现在我是处于判断-in 成功之后那条线进入的一个函数

这里的 call __alloca_probe 就是函数初始化栈的东西，这个我们可以不用管

这个写简单点就是 sub_4025B0(EAX, 400)(这个函数就是那个__split 什么那个函数)

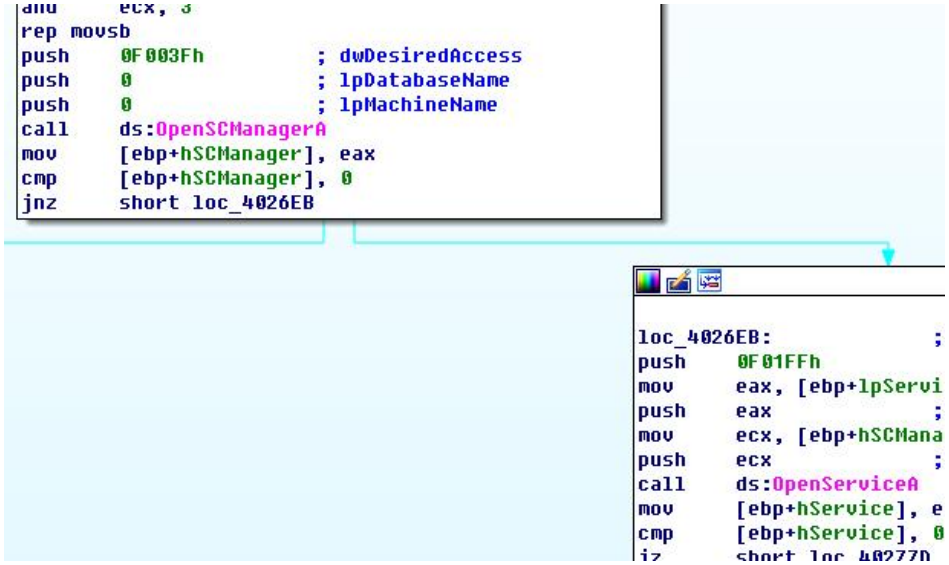
这里函数就有这么几个入参，执行看看，然后这时候的 EAX 返回值就是 0 了

这里的 eax=0，然后 test 之后 ZF=1，然后 JZ 就绿线跳转了，红线是结束这个函数并返回 1

绿线之后就是来到这里

这里显示是的计算[EDI]的字符长度也就是上面那个字符串的长度

这么一串指令到最后的



```
mov     ecx, 3
rep movsb
push    0F003Fh         ; dwDesiredAccess
push    0               ; lpDatabaseName
push    0               ; lpMachineName
call    ds:OpenSCManagerA
mov     [ebp+hSCManager], eax
cmp     [ebp+hSCManager], 0
jnz     short loc_4026EB
```

```
loc_4026EB:             ;
push    0F01FFh
mov     eax, [ebp+lpServi
push    eax             ;
mov     ecx, [ebp+hSCMana
push    ecx             ;
call    ds:OpenServiceA
mov     [ebp+hService], e
cmp     [ebp+hService], 0
jz      short loc_40277D
```

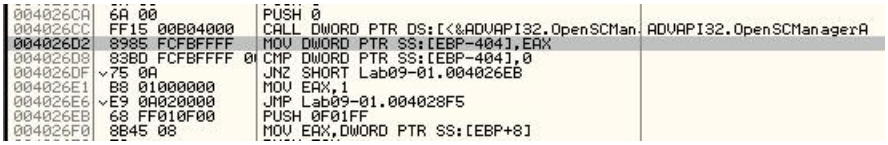
都是为了调用 OpenSCManagerA 这个函数，这个函数是在指定的计算机上建立与服务控制管理器的连接，并打开指定的服务控制管理器数据库。

前两个参数为 0 说明这个服务控制器是在本地上，这时候的寄存器。

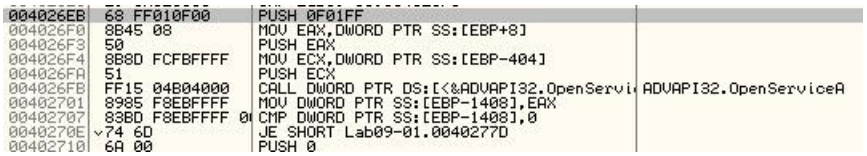
可以看出上面那个操作就是把这个字符串拼接成 EDX 所示的样子

然后这里将返回值放在[EBP-404]里面

然后和 0 比较，如果返回值 EAX 等于 0，ZF=1，JNZ 不跳转，继续执行就是结束并返回 1



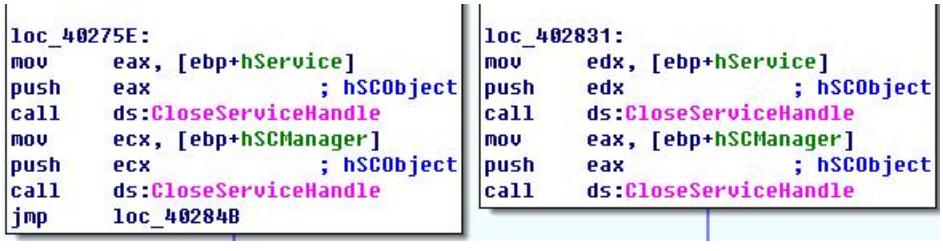
之后去到 004026EB。



这里的第一个 push 入栈的 eax 在 IDA 中标注是 lpServiceName,然后第二个 push 入栈的 ecx 是 hSCManager。这里的 eax 的值是 Lab09-01，也就是要打开的服务的名称，如果调用成功，返回一个指针，然后我们这里是第一次调用这个函数，所以这个函数会返回 0 作为返回值，和 0 执行 cmp 之后 ZF=1，则 JE 跳转。

然后这个函数就是创建一个叫 Lab09-01 的服务

这里调用成功之后，返回值肯定不为 0，所以和 0 执行 cmp 之后，ZF=0，之后 JNZ 肯定就会跳转了



跳转之后的绿线就是执行一写关闭操作，来关闭打开的 Handle 们

之后就是调用了

这里把%SYSTEMROOT%这个变量替换成了环境变量然后就成了 0012DF44 这个地址上的字符串，调用成功之后继续往下走

```
loc_40284B:             ; nSize
push    400h
lea     ecx, [ebp+BinaryPathName]
push    ecx             ; lpDst
lea     edx, [ebp+Src]
push    edx             ; lpSrc
call    ds:ExpandEnvironmentStringsA
test    eax, eax
jnz     short loc_402872
```

然后执行成功之后继续往下，这里两个入参，edx 和 ecx，lpNewFileName 的值是 ecx，然后 lpExistingFileName 也就是 edx 也是上面我们刚刚的返回值，这里要将这个可执行文件赋值到 system32 下面。执行成功之后就会跳到这里

```
loc_402891:             ; bFailIfExists
push    0
lea     ecx, [ebp+BinaryPathName]
push    ecx             ; lpNewFileName
lea     edx, [ebp+Filename]
push    edx             ; lpExistingFileName
call    ds:CopyFileA
test    eax, eax
jnz     short loc_4028B2
```

```
loc_4028B2:
lea     eax, [ebp+BinaryPathName]
push    eax             ; LPCSTR
call    sub_4015B0
add     esp, 4
test    eax, eax
jz      short loc_4028CC
```

这个函数在 MSDN 里面的解释就是检索系统目录的路径。 系统目录包含系统文件，如动态链接库和驱动程序。如果函数返回成功了，就会跳转到绿线继续执行，否则返回 1。

然后就调用了 sub_4014E0。

```
EAX 00000000
ECX 0018D7D1
EDX 0000004B
EBX 77EB0820 ntdll.77EB0820
ESP 0018E3EC
EBP 0018E4BC
ESI 00000000
EDI 00000001
EIP 77DEC2BC ntdll.77DEC2BC
```

```
; int __cdecl sub_4014E0(LPCSTR, LPCSTR lpFileName)
sub_4014E0 proc near

hFile= dword ptr -1Ch
LastAccessTime= _FILETIME ptr -18h
LastWriteTime= _FILETIME ptr -10h
CreationTime= _FILETIME ptr -8
arg_0= dword ptr  8
lpFileName= dword ptr  0Ch

push    ebp
mov     ebp, esp
sub     esp, 1Ch
push    0               ; hTemplateFile
push    80h             ; dwFlagsAndAttributes
push    3               ; dwCreationDisposition
push    0               ; lpSecurityAttributes
push    1               ; dwShareMode
push    80000000h       ; dwDesiredAccess
mov     eax, [ebp+lpFileName]
push    eax             ; lpFileName
call    ds:CreateFileA
mov     [ebp+hFile], eax
cmp     [ebp+hFile], 0
jnz     short loc_401515
```

我们还是按照先看入参然后看返回值的做法看看，这个函数的入参是 eax，这个的值是

这也就是创建这个文件叫 C:\WINDOWS\system32\kernel32.dll

这里我们执行后会发现返回值不是 0，然后就是和 0 就行 cmp，ZF=0，则 JNZ 跳转，走绿线，也就是这里

```
loc_401515:
lea     ecx, [ebp+LastWriteTime]
push    ecx             ; lpLastWriteTime
lea     edx, [ebp+LastAccessTime]
push    edx             ; lpLastAccessTime
lea     eax, [ebp+CreationTime]
push    eax             ; lpCreationTime
mov     ecx, [ebp+hFile]
push    ecx             ; hFile
call    ds:GetFileTime
test    eax, eax
jnz     short loc_401540
```

hfile 的值是 5c 也就是上面那个函数的返回值

然后这个函数会把返回值放到 lpCreationTime 和 lpLastAccessTime、lpLastWriteTime 这里

我们看看 CreationTime 的返回值好像是乱码的东西，调用成功之后就会调用这个。

```
loc_401540:
mov     eax, [ebp+hFile]
push    eax             ; hObject
call    ds:CloseHandle
push    0               ; hTemplateFile
push    80h             ; dwFlagsAndAttributes
push    3               ; dwCreationDisposition
push    0               ; lpSecurityAttributes
push    2               ; dwShareMode
push    40000000h       ; dwDesiredAccess
mov     ecx, [ebp+arg_0]
push    ecx             ; lpFileName
call    ds:CreateFileA
mov     [ebp+hFile], eax
lea     edx, [ebp+LastWriteTime]
push    edx             ; lpLastWriteTime
lea     eax, [ebp+LastAccessTime]
push    eax             ; lpLastAccessTime
lea     ecx, [ebp+CreationTime]
push    ecx             ; lpCreationTime
mov     edx, [ebp+hFile]
push    edx             ; hFile
call    ds:SetFileTime
test    eax, eax
jnz     short loc_401594
```

然后设置这个文件的时间和 kernel32.dll 的一样

然后就是关闭这个 Handle 然后返回，这个函数成功返回 0，失败返回 1

然后接着上面一层这个函数也会返回，从这里返回之后就会来到这里

然后这里有个函数 sub_4025B0，进去看看就会发现这个

```
; int __cdecl sub_4025B0(char *)
sub_4025B0 proc near

Filename= byte ptr -400h
arg_0= dword ptr  8

push    ebp
mov     ebp, esp
sub     esp, 400h
push    400h            ; nSize
lea     eax, [ebp+Filename]
push    eax             ; lpFilename
push    0               ; hModule
call    ds:GetModuleFileNameA
test    eax, eax
jnz     short loc_4025D8

loc_4025D8:             ; char *
push    0
mov     ecx, [ebp+arg_0]
push    ecx             ; char *
push    0               ; char *
push    0               ; char *
lea     edx, [ebp+Filename]
push    edx             ; char *
call    __splitpath
add     esp, 14h
xor     eax, eax
```

然后就开始退出这个函数了，分离出这个值之后返回

2.这个恶意代码的命令行选项是什么？它要求的密码是什么？

解答: 这个代码分析太长时间了哈哈哈，密码是 abcd

3.如何利用 OllyDbg 永久修补这个恶意代码，使其不需要指定的命令行密码？

解答: 修改特定的地址上的代码，然后不跳转就 ok

4.这个恶意代码基于系统的特征是什么？

解答: 恶意代码创建了一个注册表项，然后一个名为 XYZ 的服务

5.这个恶意代码通过网络执行了哪些不同操作？

解答: SLEEP, UPLOAD, DOWNLOAD, CMD，NOTHING 之类的指令

6.这个恶意代码是否有网络特征？

解答: 有，对对应网址的资源有个一个 GET 请求

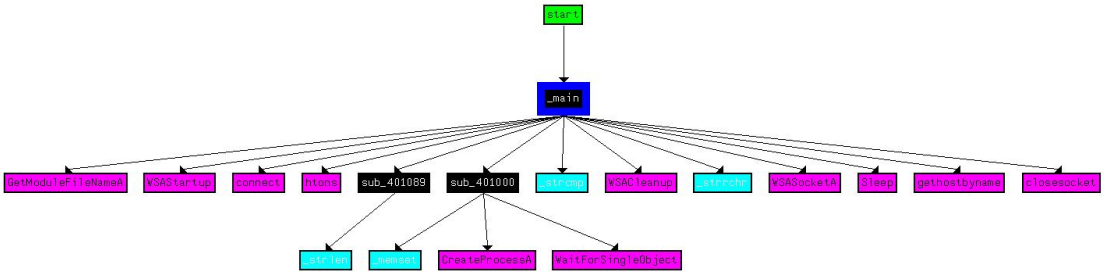
Lab09-02.exe

问题一：字符串由 strings 得到

结合 api 调用行为图，基本上可以确定是一个远控 shell 后门:

2.接下来我们运行后用 process monitor 对进程进行监控。

也并没有发现发生了什么现象。



问题 3：

把 Lab09-02.exe 载入到 ida 和 OD。

在 ida 中发现 main 的地址是 00401128，让后让 OD 跳转到此地址：

首先看到了一大串的 mov 指令

```
00401132  57              PUSH EDI
00401133  C685 50FEFFFF 3 MOV BYTE PTR SS:[EBP-1B0],31
0040113A  C685 51FEFFFF 7 MOV BYTE PTR SS:[EBP-1AF],71
00401141  C685 52FEFFFF 6 MOV BYTE PTR SS:[EBP-1AE],61
00401148  C685 53FEFFFF 7 MOV BYTE PTR SS:[EBP-1AD],7A
0040114F  C685 54FEFFFF 3 MOV BYTE PTR SS:[EBP-1AC],32
00401156  C685 55FEFFFF 7 MOV BYTE PTR SS:[EBP-1AB],77
0040115D  C685 56FEFFFF 7 MOV BYTE PTR SS:[EBP-1AA],73
00401164  C685 57FEFFFF 7 MOV BYTE PTR SS:[EBP-1A9],78
0040116B  C685 58FEFFFF 3 MOV BYTE PTR SS:[EBP-1A8],33
00401172  C685 59FEFFFF 6 MOV BYTE PTR SS:[EBP-1A7],65
00401179  C685 5AFEFFFF 6 MOV BYTE PTR SS:[EBP-1A6],64
00401180  C685 5BFEFFFF 6 MOV BYTE PTR SS:[EBP-1A5],63
00401187  C685 5CFEFFFF 0 MOV BYTE PTR SS:[EBP-1A4],0
0040118E  C685 60FEFFFF 6 MOV BYTE PTR SS:[EBP-1A0],6F
00401195  C685 61FEFFFF 6 MOV BYTE PTR SS:[EBP-19F],63
0040119C  C685 62FEFFFF 6 MOV BYTE PTR SS:[EBP-19E],6C
004011A3  C685 63FEFFFF 2 MOV BYTE PTR SS:[EBP-19D],2E
004011AA  C685 64FEFFFF 6 MOV BYTE PTR SS:[EBP-19C],65
004011B1  C685 65FEFFFF 7 MOV BYTE PTR SS:[EBP-19B],78
004011B8  C685 66FEFFFF 6 MOV BYTE PTR SS:[EBP-19A],65
004011BF  C685 67FEFFFF 0 MOV BYTE PTR SS:[EBP-199],0
004011C6  B9 08000000     MOV ECX,8
004011CB  BE 34504000     MOV ESI,Lab09-02.00405034
004011D0  8DBD 10FEFFFF   LEA EDI,DWORD PTR SS:[EBP-1F0]
004011D6  F3:A5           REP MOVS DWORD PTR ES:[EDI],DWORD PTR DS
004011D8  A4              MOVS BYTE PTR ES:[EDI],BYTE PTR DS:[ESI
004011D9  C785 48FEFFFF 0 MOV DWORD PTR SS:[EBP-1B8],0
004011E3  C685 00FDFFFF 0 MOV BYTE PTR SS:[EBP-300],0
004011EA  B9 43000000     MOV ECX,43
004011EF  33C0            XOR EAX,EAX
004011F1  8DBD 01FDFFFF   LEA EDI,DWORD PTR SS:[EBP-2FF]
004011F7  F3:AB           REP STOS DWORD PTR ES:[EDI]
004011F9  AA              STOS BYTE PTR ES:[EDI]
004011FA  68 0E010000     PUSH 10E
```

这是将字节移动到栈中的操作，出现了两个 0，很明显是终止符。把这一长串的字符分成两部分。查看这些字符到底都是什么可以在 ida 中直接快捷键 R 键完成转换。

第一串并不能看出什么，第二串字符有可能是一个文件名。继续向下分析：

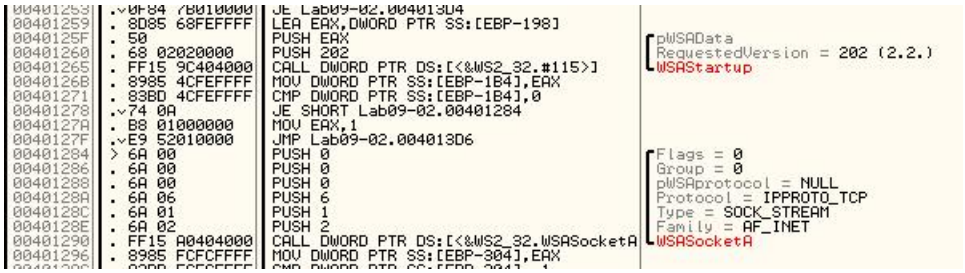
当看到 GetModleFileName 时停一下，这是用于获取当前运行的完整路径，F8 就看到确实出现了完整路径。

```
lea     edi, [ebp+var_1F0]
rep movsd
movsb
mov     [ebp+var_1B8], 0
mov     [ebp+Filename], 0
mov     ecx, 43h
xor     eax, eax
lea     edi, [ebp+var_2FF]
rep stosd
stosb
push    10Eh            ; nSize
lea     eax, [ebp+Filename]
push    eax             ; lpFilename
push    0               ; hModule
call    ds:GetModuleFileNameA
push    5Ch             ; int
lea     ecx, [ebp+Filename]
push    ecx             ; char *
call    _strrchr
add     esp, 8
mov     [ebp+var_4], eax
mov     edx, [ebp+var_4]
add     edx, 1
mov     [ebp+var_4], edx
mov     eax, [ebp+var_4]
push    eax             ; char *
lea     ecx, [ebp+var_1A0]
```

```
004011F1  8DBD 01FDFFFF   LEA EDI,DWORD PTR SS:[EBP-2FF]
004011F7  F3:AB           REP STOS DWORD PTR ES:[EDI]
004011F9  AA              STOS BYTE PTR ES:[EDI]
004011FA  68 0E010000     PUSH 10E
004011FF  8D85 00FDFFFF   LEA EAX,DWORD PTR SS:[EBP-300]
00401205  50              PUSH EAX
00401206  6A 00           PUSH 0
00401208  FF15 0C404000   CALL DWORD PTR DS:[<&KERNEL32.GetModule KERNEL32.GetModuleFileNameA
0040120E  6A 5C           PUSH 5C
00401210  8D8D 00FDFFFF   LEA ECX,DWORD PTR SS:[EBP-300]
00401216  51              PUSH ECX
00401217  E8 34030000     CALL Lab09-02.00401550
0040121C  83C4 08         ADD ESP,8
0040121F  8945 FC         MOV DWORD PTR SS:[EBP-4],EAX
00401222  8B55 FC         MOV EDX,DWORD PTR SS:[EBP-4]
00401225  83C2 01         ADD EDX,1
00401228  8955 FC         MOV DWORD PTR SS:[EBP-4],EDX
0040122B  8B45 FC         MOV EAX,DWORD PTR SS:[EBP-4]
0040122E  50              PUSH EAX
0040122F  8D8D 60FEFFFF   LEA ECX,DWORD PTR SS:[EBP-1A0]
00401235  51              PUSH ECX
00401236  E8 85020000     CALL Lab09-02.004014C0
0040123B  83C4 08         ADD ESP,8
0040123E  85C0            TEST EAX,EAX
00401240  74 0A           JE SHORT Lab09-02.0040124C
00401242  B8 01000000     MOV EAX,1
00401247  E9 8A010000     JMP Lab09-02.004013D6
```

继续向下走：

这里调用了 00401550 这个函数，0401550 这个函数表示的是库函数_strrchr，这个函数就是想获取文件的名称。



这里看到调用了 004014C0，OD 没有解析出它的名称，但是我们看到了两个参数，一个是前面所怀疑的 ocl.exe，另一个就是文件名了。

```
.text:0040120E                  push    5Ch             ; int
.text:00401210                  lea     ecx, [ebp+Filename]
.text:00401216                  push    ecx             ; char *
.text:00401217                  call    _strrchr
.text:0040121C                  add     esp, 8
.text:0040121F                  mov     [ebp+var_4], eax
.text:00401222                  mov     edx, [ebp+var_4]
.text:00401225                  add     edx, 1
.text:00401228                  mov     [ebp+var_4], edx
.text:0040122B                  mov     eax, [ebp+var_4]
.text:0040122E                  push    eax             ; char *
.text:0040122F                  lea     ecx, [ebp+var_1A0]
.text:00401235                  push    ecx             ; char *
.text:00401236                  call    _strcmp
.text:0040123B                  add     esp, 8
.text:0040123E                  test    eax, eax
.text:00401240                  jz      short loc_40124C
.text:00401242                  mov     eax, 1
.text:00401247                  jmp     loc_4013D6
```

看一下 ida 就清楚了，这是一个字符串的比较函数。我们知道这两个字符串不行等，会返回一个非零值，test 操作依旧是非零值，所以呢程序也就退出了，分析到这里就很清楚了，想要让这个恶意代码运行修改文件名称就好了。这里也解释了问题 2。我们可以修改文件名为 ocl.exe，然后重新运行就可以了。

问题 4：快捷键 G 跳转到地址 401133：

然后发现，这个就是一系列的赋值操作。

```
.text:00401133                  mov     [ebp+var_1B0], 31h
.text:0040113A                  mov     [ebp+var_1AF], 71h
.text:00401141                  mov     [ebp+var_1AE], 61h
.text:00401148                  mov     [ebp+var_1AD], 7Ah
.text:0040114F                  mov     [ebp+var_1AC], 32h
.text:00401156                  mov     [ebp+var_1AB], 77h
.text:0040115D                  mov     [ebp+var_1AA], 73h
.text:00401164                  mov     [ebp+var_1A9], 78h
.text:0040116B                  mov     [ebp+var_1A8], 33h
.text:00401172                  mov     [ebp+var_1A7], 65h
.text:00401179                  mov     [ebp+var_1A6], 64h
.text:00401180                  mov     [ebp+var_1A5], 63h
.text:00401187                  mov     [ebp+var_1A4], 0
.text:0040118E                  mov     [ebp+var_1A0], 6Fh
.text:00401195                  mov     [ebp+var_19F], 63h
.text:0040119C                  mov     [ebp+var_19E], 6Ch
.text:004011A3                  mov     [ebp+var_19D], 2Eh
.text:004011AA                  mov     [ebp+var_19C], 65h
.text:004011B1                  mov     [ebp+var_19B], 78h
.text:004011B8                  mov     [ebp+var_19A], 65h
.text:004011BF                  mov     [ebp+var_199], 0
.text:004011C6                  mov     ecx, 8
```

问题 5：

在 ida 中找到这个位置，地址是 4012BD，然后在 OD 中跳转到这个地址：

```
text:004012B6                   lea     edx, [ebp+var_1B0]
text:004012BC                   push    edx             ; char *
text:004012BD                   call    sub_401089
text:004012C2                   add     esp, 8
text:004012C5                   mov     [ebp+name], eax
text:004012C8                   mov     eax, [ebp+name]
text:004012CB                   push    eax             ; name
text:004012CC                   call    ds:gethostbyname
text:004012D2                   mov     [ebp+var_1BC], eax
text:004012D8                   cmp     [ebp+var_1BC], 0
text:004012DF                   jnz     short loc_401304
text:004012E1                   mov     ecx, [ebp+s]
text:004012E7                   push    ecx             ; s
text:004012E8                   call    ds:closesocket
text:004012EE                   call    ds:WSACleanup
text:004012F4                   push    7530h           ; dwMilliseconds
text:004012F9                   call    ds:Sleep
text:004012FF                   jmp     loc_40124C
```

发现了第一个参数就是之前的第一个字符串，第二个参数是一个数据缓冲区，暂时无法识别。

```
004012B5  . 51              PUSH ECX                                 Arg2
004012B6  . 8D95 50FEFFFF  LEA EDX,DWORD PTR SS:[EBP-1B0]
004012BC  . 52              PUSH EDX                                 Arg1
004012BD  . E8 C7FDFFFF    CALL Lab09-02.00401089                   Lab09-02.00401089
004012C2  . 83C4 08         ADD ESP,8
004012C5  . 8945 F8         MOV DWORD PTR SS:[EBP-8],EAX
004012C8  . 8B45 F8         MOV EAX,DWORD PTR SS:[EBP-8]
004012CB  . 50              PUSH EAX                                 Name
004012CC  . FF15 A4404000  CALL DWORD PTR DS:[<&WS2_32.#52>]        gethostbyname
004012D2  . 8985 44FEFFFF  MOV DWORD PTR SS:[EBP-1BC],EAX
004012D8  . 83BD 44FEFFFF  CMP DWORD PTR SS:[EBP-1BC],0
004012DF  . 75 23           JNZ SHORT Lab09-02.00401304
```

问题 6:

我们进入这个 call 看一下，F5 查看 C 代码

可以看到这里有两个参数，a1 表示那个未能识别的参数，a2 表示“1qaz2wsx3edc”。下面有一个很明显的异或操作，就是将这两个字符串异或。进入 OD 进入这个 call，找到这个异或操作的位置

```
1  char *__cdecl sub_401089(char *a1, int a2)
2  {
3    signed int i; // [sp+4h] [bp-108h]@1
4    signed int v4; // [sp+8h] [bp-104h]@1
5    char v5; // [sp+Ch] [bp-100h]@1
6    char v6; // [sp+Dh] [bp-FFh]@1
7    __int16 v7; // [sp+109h] [bp-3h]@1
8    char v8; // [sp+10Bh] [bp-1h]@1
9
10   v5 = 0;
11   memset(&v6, 0, 0xFCu);
12   v7 = 0;
13   v8 = 0;
14   v4 = strlen(a1);
15   for ( i = 0; i < 32; ++i )
16     *(&v5 + i) = a1[i % v4] ^ *(_BYTE *)(i + a2);
17   return &v5;
18 }
```

```
004010F5  . 0FBE0A          MOVSX ECX,BYTE PTR DS:[EDX]
004010F8  . 8B85 F8FEFFFF  MOV EAX,DWORD PTR SS:[EBP-108]
004010FE  . 99              CDQ
004010FF  . F7BD FCFEFFFF  IDIV DWORD PTR SS:[EBP-104]
00401105  . 8B45 08        MOV EAX,DWORD PTR SS:[EBP+8]
00401108  . 0FBE1410        MOVSX EDX,BYTE PTR DS:[EAX+EDX]
0040110C  . 33CA            XOR ECX,EDX
0040110E  . 8B85 F8FEFFFF  MOV EAX,DWORD PTR SS:[EBP-108]
00401114  . 888C05 00FFFFFF MOV BYTE PTR SS:[EBP+EAX-100],CL
0040111B  .^EB B7           JMP SHORT Lab09-02.004010D4
0040111D  > 8D85 00FFFFFF  LEA EAX,DWORD PTR SS:[EBP-100]
00401123  . 5F              POP EDI
00401124  . 8BE5            MOV ESP,EBP
00401126  . 5D              POP EBP
00401127  . C3              RETN
00401128  $ 55              PUSH EBP
00401129  . 8BEC            MOV EBP,ESP
0040112B  . 81EC 04030000  SUB ESP,304
00401131  . 56              PUSH ESI
00401132  . 57              PUSH EDI
00401133  . C685 50FEFFFF  MOV BYTE PTR SS:[EBP-1B0],31
0040113A  . C685 51FEFFFF  MOV BYTE PTR SS:[EBP-1AF],71
00401141  . C685 52FEFFFF  MOV BYTE PTR SS:[EBP-1AE],61
```

这里可以看到是由 edx 和 ecx 进行异或操作，F9 运行：

这样就看到了异或操作之后出来的一个网址，这个就是恶意代码使用的域名。

问题 7：恶意代码使用了“1qaz2wsx3edc”这个字符串来混淆域名。问题 8：

```
text:00401026                push    0               ; int
text:00401028                lea     ecx, [ebp+ProcessInformation]
text:0040102B                push    ecx             ; void *
text:0040102C                call    _memset
text:00401031                add     esp, 0Ch
text:00401034                mov     [ebp+StartupInfo.dwFlags], 101h
text:0040103B                mov     [ebp+StartupInfo.wShowWindow], 0
text:00401041                mov     edx, [ebp+arg_10]
text:00401044                mov     [ebp+StartupInfo.hStdInput], edx
text:00401047                mov     eax, [ebp+StartupInfo.hStdInput]
text:0040104A                mov     [ebp+StartupInfo.hStdError], eax
text:0040104D                mov     ecx, [ebp+StartupInfo.hStdError]
text:00401050                mov     [ebp+StartupInfo.hStdOutput], ecx
text:00401053                lea     edx, [ebp+ProcessInformation]
text:00401056                push    edx             ; lpProcessInformation
text:00401057                lea     eax, [ebp+StartupInfo]
text:0040105A                push    eax             ; lpStartupInfo
text:0040105B                push    0               ; lpCurrentDirectory
text:0040105D                push    0               ; lpEnvironment
text:0040105F                push    0               ; dwCreationFlags
text:00401061                push    1               ; bInheritHandles
text:00401063                push    0               ; lpThreadAttributes
text:00401065                push    0               ; lpProcessAttributes
text:00401067                push    offset CommandLine ; "cmd"
text:0040106C                push    0               ; lpApplicationName
text:0040106E                call    ds:CreateProcessA
```

其实这是一个反向的 shell，反向 shell(Reverse shell)是一种往远程机器发送 shell 命令的技术。在 CreateProcessA 这个函数被调用之前， STARTUPINFO 结构就已经被修改了，这个结构用于指定新进程的主窗口特性。首先可以看到，StartupInfo.wShowWindow 的值被设置为了 0，表示它会以窗口隐藏的方式运行，而运行的对象就是 cmd.exe，所以受害用户是看不到程序运行的，也就是我们在运行时并不会看到什么。接着我们可以看到 STARTUPINFO 结构中的标准流被设置为一个套接字，这也就直接绑定了套接字和 cmd.exe 的标准流，所以 cmd.exe 被启动后，所有经过套接字的数据都将发送到 cmd.exe，并且 cmd.exe 产生的所有输出都将通过套接字发出。

Lab09-03.exe

pFile	Data	Description	Value
00005000	000055EC	Hint/Name RVA	0000 DLL1Print
00005004	00000000	End of Imports	DLL1.dll
00005008	00005602	Hint/Name RVA	0001 DLL2ReturnJ
0000500C	00005610	Hint/Name RVA	0000 DLL2Print
00005010	00000000	End of Imports	DLL2.dll
00005014	000055B0	Hint/Name RVA	02DF WriteFile
00005018	0000585E	Hint/Name RVA	01C0 LCMapStringW
0000501C	000055A2	Hint/Name RVA	001B CloseHandle
00005020	00005592	Hint/Name RVA	01C2 LoadLibraryA
00005024	00005580	Hint/Name RVA	013E GetProcAddress
00005028	0000586E	Hint/Name RVA	0153 GetStringTypeA
0000502C	00005578	Hint/Name RVA	0296 Sleep
00005030	00005626	Hint/Name RVA	00CA GetCommandLineA
00005034	00005638	Hint/Name RVA	0174 GetVersion
00005038	00005646	Hint/Name RVA	007D ExitProcess
0000503C	00005654	Hint/Name RVA	029E TerminateProcess
00005040	00005668	Hint/Name RVA	00F7 GetCurrentProcess
00005044	0000567C	Hint/Name RVA	02AD UnhandledExceptionFilter
00005048	00005698	Hint/Name RVA	0124 GetModuleFileNameA
0000504C	000056AE	Hint/Name RVA	00B2 FreeEnvironmentStringsA
00005050	000056C8	Hint/Name RVA	00B3 FreeEnvironmentStringsW
00005054	000056E2	Hint/Name RVA	02D2 WideCharToMultiByte
00005058	000056F8	Hint/Name RVA	0106 GetEnvironmentStrings
0000505C	00005710	Hint/Name RVA	0108 GetEnvironmentStringsW
00005060	0000572A	Hint/Name RVA	026D SetHandleCount
00005064	0000573C	Hint/Name RVA	0152 GetStdHandle
00005068	0000574C	Hint/Name RVA	0115 GetFileType
0000506C	0000575A	Hint/Name RVA	0150 GetStartupInfoA
00005070	0000576C	Hint/Name RVA	0126 GetModuleHandleA
00005074	00005780	Hint/Name RVA	0109 GetEnvironmentVariableA
00005078	0000579A	Hint/Name RVA	0175 GetVersionExA
0000507C	000057AA	Hint/Name RVA	019D HeapDestroy
00005080	000057B8	Hint/Name RVA	019B HeapCreate
00005084	000057C6	Hint/Name RVA	02BF VirtualFree
00005088	000057D4	Hint/Name RVA	019F HeapFree
0000508C	000057E0	Hint/Name RVA	022F RtlUnwind
00005090	000057EC	Hint/Name RVA	0199 HeapAlloc
00005094	000057F8	Hint/Name RVA	00BF GetCPInfo
00005098	00005804	Hint/Name RVA	00B9 GetACP

问题 1：

首先用 peview 查看导入表：

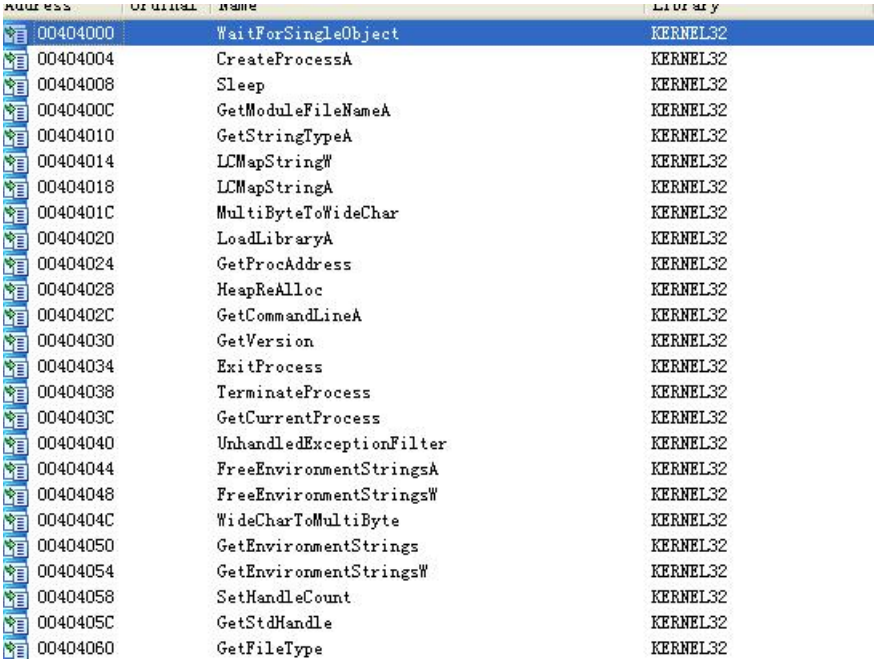
通过 peview 可以看到，分别导入了以下四个 dll。

还有些 dll 可能在程序运行时加载，需要 ida 分析，一般会需要 loadlibrary

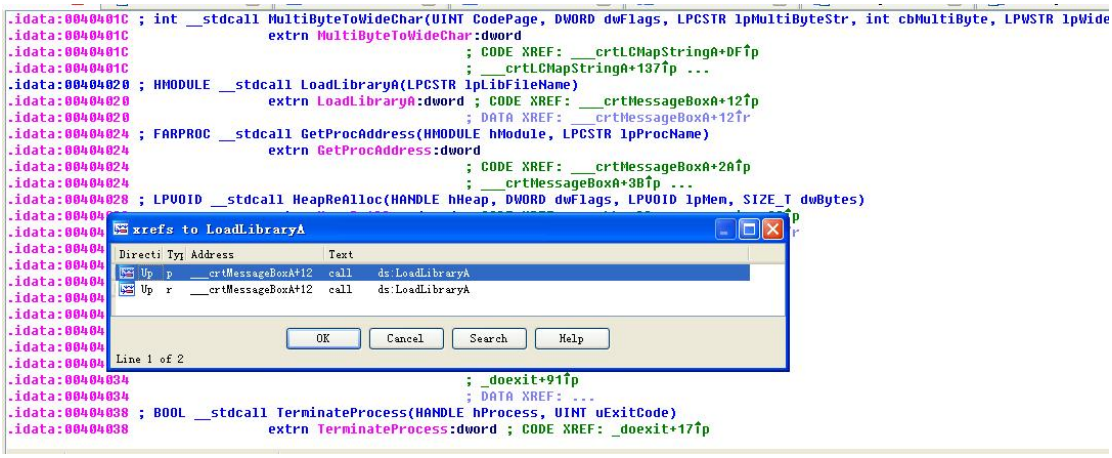
Q1.lab09-03.exe 导入了哪些 dll

A1.导入的 dll 有：kernel32.dll、NetAPI32.dll、DLLI.dll 和 DLL2.dl

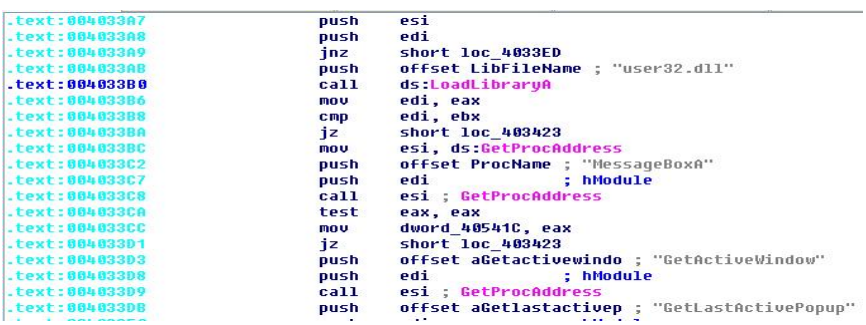
在 imports 窗口找到 loadlibrary



Address	Ordinal	Name	Library
00404000		WaitForSingleObject	KERNEL32
00404004		CreateProcessA	KERNEL32
00404008		Sleep	KERNEL32
0040400C		GetModuleFileNameA	KERNEL32
00404010		GetStringTypeA	KERNEL32
00404014		LCMapStringW	KERNEL32
00404018		LCMapStringA	KERNEL32
0040401C		MultiByteToWideChar	KERNEL32
00404020		LoadLibraryA	KERNEL32
00404024		GetProcAddress	KERNEL32
00404028		HeapReAlloc	KERNEL32
0040402C		GetCommandLineA	KERNEL32
00404030		GetVersion	KERNEL32
00404034		ExitProcess	KERNEL32
00404038		TerminateProcess	KERNEL32
0040403C		GetCurrentProcess	KERNEL32
00404040		UnhandledExceptionFilter	KERNEL32
00404044		FreeEnvironmentStringsA	KERNEL32
00404048		FreeEnvironmentStringsW	KERNEL32
0040404C		WideCharToMultiByte	KERNEL32
00404050		GetEnvironmentStrings	KERNEL32
00404054		GetEnvironmentStringsW	KERNEL32
00404058		SetHandleCount	KERNEL32
0040405C		GetStdHandle	KERNEL32
00404060		GetFileType	KERNEL32



p 表示函数调用，r 表示读取，我们要关注的调用，先看第一个 p，可以看到



```
.text:004033A7                 push    esi
.text:004033A8                 push    edi
.text:004033A9                 jnz     short loc_4033ED
.text:004033AB                 push    offset LibFileName ; "user32.dll"
.text:004033B0                 call    ds:LoadLibraryA
.text:004033B6                 mov     edi, eax
.text:004033B8                 cmp     edi, ebx
.text:004033BA                 jz      short loc_403423
.text:004033BC                 mov     esi, ds:GetProcAddress
.text:004033C2                 push    offset ProcName ; "MessageBoxA"
.text:004033C7                 push    edi             ; hModule
.text:004033C8                 call    esi ; GetProcAddress
.text:004033CA                 test    eax, eax
.text:004033CC                 mov     dword_40541C, eax
.text:004033D1                 jz      short loc_403423
.text:004033D3                 push    offset aGetactivewindo ; "GetActiveWindow"
.text:004033D8                 push    edi             ; hModule
.text:004033D9                 call    esi ; GetProcAddress
.text:004033DB                 push    offset aGetlastactivep ; "GetLastActivePopup"
```

在调用时其第一个参数是 dll3.dll，再看一下第二个

```
.text:004033A7                  push    esi
.text:004033A8                  push    edi
.text:004033A9                  jnz     short loc_4033ED
.text:004033AB                  push    offset LibFileName ; "user32.dll"
.text:004033B0                  call    ds:LoadLibraryA
.text:004033B6                  mov     edi, eax
.text:004033B8                  cmp     edi, ebx
.text:004033BA                  jz      short loc_403423
.text:004033BC                  mov     esi, ds:GetProcAddress
.text:004033C2                  push    offset ProcName ; "MessageBoxA"
.text:004033C7                  push    edi             ; hModule
.text:004033C8                  call    esi ; GetProcAddress
.text:004033CA                  test    eax, eax
.text:004033CC                  mov     dword_40541C, eax
.text:004033D1                  jz      short loc_403423
.text:004033D3                  push    offset aGetactivewindo ; "GetActiv
.text:004033D8                  push    edi             ; hModule
.text:004033D9                  call    esi ; GetProcAddress
.text:004033DB                  push    offset aGetlastactivep ; "GetLastA
.text:004033E0                  push    edi             ; hModule
.text:004033E1                  mov     dword_405420, eax
```

其参数是 use32.dll，那么也就是说程序其实一共导入了这六个 dll

再看第二个问题，peview 载入 dll1.dll，映像基地址是 0x10000000

```
.text:004033A7                  push    esi
.text:004033A8                  push    edi
.text:004033A9                  jnz     short loc_4033ED
.text:004033AB                  push    offset LibFileName ; "user32.dll"
.text:004033B0                  call    ds:LoadLibraryA
.text:004033B6                  mov     edi, eax
.text:004033B8                  cmp     edi, ebx
.text:004033BA                  jz      short loc_403423
.text:004033BC                  mov     esi, ds:GetProcAddress
.text:004033C2                  push    offset ProcName ; "MessageBoxA"
.text:004033C7                  push    edi             ; hModule
.text:004033C8                  call    esi ; GetProcAddress
.text:004033CA                  test    eax, eax
.text:004033CC                  mov     dword_40541C, eax
.text:004033D1                  jz      short loc_403423
.text:004033D3                  push    offset aGetactivewindo ; "GetActiveWindow"
.text:004033D8                  push    edi             ; hModule
.text:004033D9                  call    esi ; GetProcAddress
.text:004033DB                  push    offset aGetlastactivep ; "GetLastActivePopup"
.text:004033E0                  push    edi             ; hModule
.text:004033E1                  mov     dword_405420, eax
.text:004033E6                  call    esi ; GetProcAddress
.text:004033E8                  mov     dword_405424, eax
```

dll2，dll3 都是如此

Q2.dll1.dll,dll2.dll,dll3.dll 要求的基地址是多少

A2.这三个 DLL 都要求相同的基准地址: 0x 10000000

dll1 地址为 10000000，dll2 地址为 001d0000，这个地址是动态的，可能再做一次，或者在不同系统上就不一样了，而 dll3.dll 是程序中利用 loadLibrary 动态加载的，找到地址。

在 od 中跳到 401041, 然后在其下一行下断点，执行，在点击 M

```
0040102C | . E8 AF030000    CALL Lab09-02.004013E0
00401031 | . 83C4 0C        ADD ESP,0C
00401034 | . C745 D4 01010( MOV DWORD PTR SS:[EBP-2C],101
0040103B | . 66:C745 D8 00( MOV WORD PTR SS:[EBP-28],0
00401041 | . 8B55 18        MOV EDX,DWORD PTR SS:[EBP+18]
00401044 | . 8955 E0        MOV DWORD PTR SS:[EBP-20],EDX
00401047 | . 8B45 E0        MOV EAX,DWORD PTR SS:[EBP-20]
0040104A | . 8945 E8        MOV DWORD PTR SS:[EBP-18],EAX
0040104D | . 8B4D E8        MOV ECX,DWORD PTR SS:[EBP-18]
00401050 | . 894D E4        MOV DWORD PTR SS:[EBP-1C],ECX
00401053 | . 8D55 F0        LEA EDX,DWORD PTR SS:[EBP-10]
00401056 | . 52             PUSH EDX                          pProcessInfo
00401057 | . 8D45 A8        LEA EAX,DWORD PTR SS:[EBP-58]
0040105A | . 50             PUSH EAX                          pStartupInfo
0040105B | . 6A 00           PUSH 0                            CurrentDir = NULL
0040105D | . 6A 00           PUSH 0                            pEnvironment = NULL
0040105F | . 6A 00           PUSH 0                            CreationFlags = 0
00401061 | . 6A 01           PUSH 1                            InheritHandles = TRUE
00401063 | . 6A 00           PUSH 0                            pThreadSecurity = NULL
00401065 | . 6A 00           PUSH 0                            pProcessSecurity = NULL
00401067 | . 68 30504000     PUSH Lab09-02.00405030            CommandLine = "cmd"
0040106C | . 6A 00           PUSH 0                            ModuleFileName = NULL
0040106E | . FF15 04404000   CALL DWORD PTR DS:[<&KERNEL32.CreatePro  CreateProcessA
00401074 | . 8945 EC         MOV DWORD PTR SS:[EBP-14],EAX
00401077 | . 6A FF           PUSH -1                           Timeout = INFINITE
```

此时就可以看到 dll3 的地址是 001f0000

Q3.当使用 Ollydbg 调试 Lab09-03.exe 时，为 dll1.dll,dll2.dll,dll3.dll 分配的基地址是什么

A3.dll1 地址为 10000000，dll2 地址为 001d0000，dll3.dll 地址是 001f0000。

为了回答第 4 个问题:ida 载入 dll1.dll,找到 dll1print

```
.text:10001020 ; Exported entry   1. DLL1Print
.text:10001020
.text:10001020 ; =============== S U B R O U T I N E =======================================
.text:10001020
.text:10001020 ; Attributes: bp-based frame
.text:10001020
.text:10001020                 public DLL1Print
.text:10001020 DLL1Print       proc near               ; DATA XREF: .rdata:off_10007BA8↓o
.text:10001020                 push    ebp
.text:10001021                 mov     ebp, esp
.text:10001023                 mov     eax, dword_10008030
.text:10001028                 push    eax
.text:10001029                 push    offset aDll1MysteryDat ; "DLL 1 mystery data %d\n"
.text:1000102E                 call    sub_10001038
.text:10001033                 add     esp, 8
.text:10001036                 pop     ebp
.text:10001037                 retn
.text:10001037 DLL1Print       endp
.text:10001037
```

Q4.当 lab09-03.exe 调用 dll1.dll 中的一个导入函数时，这个导入函数都做了什么

A4.DLL1Print 被调用，它打印出“DLL 1 mystery data”，随后是一一个全局变量的内容

调用了 sub_10001038,跟进去

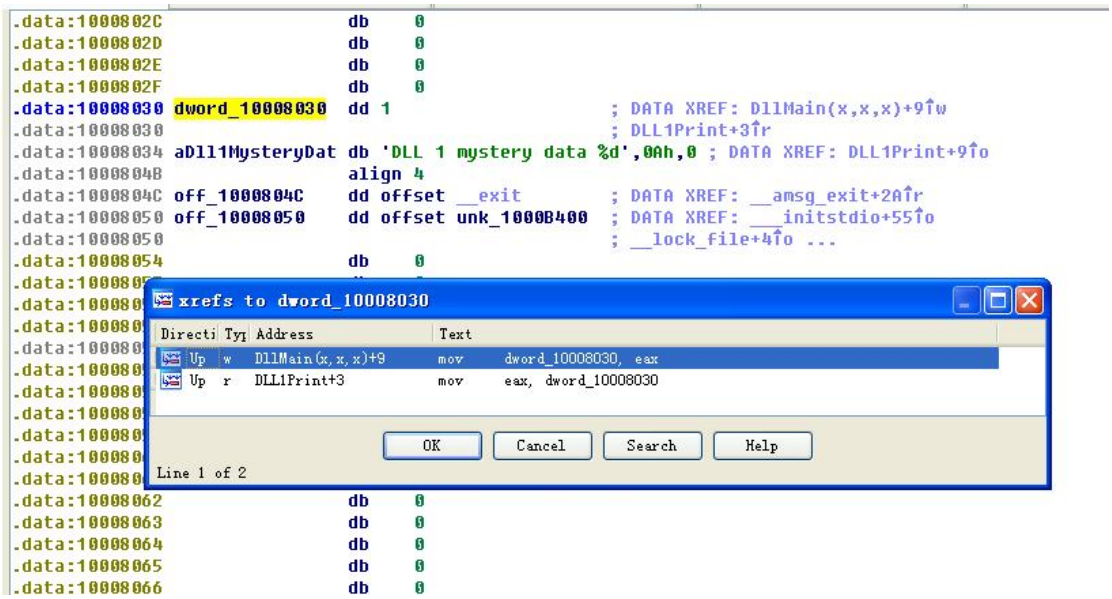
```
.text:10001038
.text:10001038 arg_0           = dword ptr  4
.text:10001038 arg_4           = dword ptr  8
.text:10001038
.text:10001038                 push    ebx
.text:10001039                 push    esi
.text:1000103A                 mov     esi, offset stru_10008070
.text:1000103F                 push    edi
.text:10001040                 push    esi
.text:10001041                 push    1
.text:10001043                 call    __lock_file2
.text:10001048                 push    esi
.text:10001049                 call    __stbuf
.text:1000104E                 mov     edi, eax
.text:10001050                 lea     eax, [esp+18h+arg_4]
.text:10001054                 push    eax             ; int
.text:10001055                 push    [esp+1Ch+arg_0] ; int
.text:10001059                 push    esi             ; FILE *
.text:1000105A                 call    sub_10001439
.text:1000105F                 push    esi
.text:10001060                 push    edi
.text:10001061                 mov     ebx, eax
.text:10001063                 call    __ftbuf
.text:10001068                 push    esi
.text:10001069                 push    1
.text:1000106B                 call    __unlock_file2
.text:10001070                 add     esp, 28h
```

```
sub_10001038 proc near

arg_0= dword ptr  4
arg_4= dword ptr  8

push    ebx
push    esi
mov     esi, offset stru_10008070
push    edi
push    esi
push    1
call    __lock_file2
push    esi
call    __stbuf
mov     edi, eax
lea     eax, [esp+18h+arg_4]
push    eax             ; int
push    [esp+1Ch+arg_0] ; int
push    esi             ; FILE *
call    sub_10001439
push    esi
push    edi
mov     ebx, eax
call    __ftbuf
push    esi
push    1
call    __unlock_file2
add     esp, 28h
mov     eax, ebx
pop     edi
pop     esi
pop     ebx
retn
sub_10001038 endp
```

可以看到这个函数是有两个参数,第一个参数是字符串,字符串里有%d,\n。

这种我们之前分析过，其实 sub_10001038 是 printf,第二个参数在 eax 中

往上看，可知来自 dword_10008030,双击查看



然后 ctrl+x 查看交叉引用,可以看到一个是 w 一个是 r,关注的是值被改写时是什么情况，所以双击查看第一个。

```
hinstDLL= dword ptr  8
fdwReason= dword ptr  0Ch
lpvReserved= dword ptr  10h

push    ebp
mov     ebp, esp
call    ds:GetCurrentProcessId
mov     dword_10008030, eax
mov     al, 1
pop     ebp
retn    0Ch
_DllMain@12 endp
```

```
; Exported entry   1. DLL1Print


; Attributes: bp-based frame

public DLL1Print
DLL1Print proc near
push    ebp
mov     ebp, esp
mov     eax, dword_10008030
push    eax
push    offset aDll1MysteryDat ; "DLL 1 mystery data %d\r
call    sub_10001038
add     esp, 8
pop     ebp
retn
DLL1Print endp
```

可以看到它保存的其实就是 GetCurrentProcessId 的返回值，也就是当前进程的 id 值。找到了 writefile，看到第一个参数是 hfile,保存的是要写入的文件句柄，而第二个参数是要写入的内容，也就是 malware….字符串，为了知道文件名称，所以需要分析的是第一个参数，可以看到它其实是 dll2returnj 的返回值。所以需要 ida 分析 dll2.dll，先看一下 dll2print

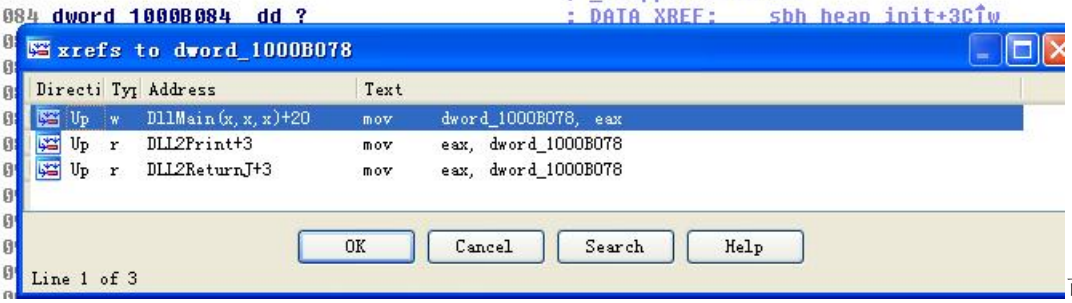
这里同样有打印的操作，要打印的数据保存是在 eax，来自 dword_1000b078

双击后查看交叉引用，如上图所示

分析 w

```
push    ebp
mov     ebp, esp
push    0               ; hTemplateFile
push    80h             ; dwFlagsAndAttributes
push    2               ; dwCreationDisposition
push    0               ; lpSecurityAttributes
push    0               ; dwShareMode
push    40000000h       ; dwDesiredAccess
push    offset FileName ; "temp.txt"
call    ds:CreateFileA
mov     dword_1000B078, eax
mov     al, 1
pop     ebp
retn    0Ch
_DllMain@12 endp
```

```
078 dword_1000B078  dd ?                    ; DATA XREF: DllMain(x,x,x)+20↑w
078                                         ; DLL2Print+3↑r ...
07C dword_1000B07C  dd ?                    ; DATA XREF: _wctomb+1A↑r
07C                                         ; _toupper+2C↑r ...
080 ; volatile LONG Addend
080 Addend          dd ?                    ; DATA XREF: _wctomb+5↑o
080                                         ; _toupper+1F↑o ...
084 dword_1000B084  dd ?                    ; DATA XREF:    sbh_heap_init+3C↑w
```



可以看到之前是调用了 createfile，打开 temp.txt 的句柄就保存在 dword_1000b078

前面打印函数打印也就是这个句柄

接下来分析 dll2returnj

```
; Exported entry   2. DLL2ReturnJ


; Attributes: bp-based frame

public DLL2ReturnJ
DLL2ReturnJ proc near
push    ebp
mov     ebp, esp
mov     eax, dword_1000B078
pop     ebp
retn
DLL2ReturnJ endp
```

主要就是讲 dword 也就是 temp.txt 的句柄赋给 eax，之后就返回了

可以看到，dll2reurnj 返回的 eax 就是 temp.txt 文件的句柄

那么后面的 writefile 函数写入的文件也就是 temp.txt

Q5.当 lab09-03.exe 调用 WriteFile 函数时，写入的文件名是什么

A5.写入的文件名是 temp.txt

回到第 6 个问题,定位到相应位置

```
text:10001070                   public DLL3Print
text:10001070 DLL3Print         proc near               ; DATA XREF: .rdata:off_10007B88↓o
text:10001070                   push    ebp
text:10001071                   mov     ebp, esp
text:10001073                   push    offset WideCharStr
text:10001078                   push    offset aDll3MysteryDat ; "DLL 3 mystery data %d\n"
text:1000107D                   call    sub_10001087
text:10001082                   add     esp, 8
text:10001085                   pop     ebp
text:10001086                   retn
text:10001086 DLL3Print         endp
text:10001086
text:10001087
text:10001087 ; =============== S U B R O U T I N E =======================================
text:10001087
text:10001087
text:10001087 sub_10001087      proc near               ; CODE XREF: DLL3Print+D↑p
text:10001087
text:10001087 arg_0             = dword ptr  4
text:10001087 arg_4             = dword ptr  8
text:10001087
text:10001087                   push    ebx
text:10001088                   push    esi
text:10001089                   mov     esi, offset stru_10008098
text:1000108E                   push    edi
```

在调用它之前看看发生了什么,先是加载了 dll3.dll 到内存中

之后是调用 getprocaddress 获取 dll3.dll 中的 dll3print 的地址

之后 call [ebp+var_8]来执行该函数

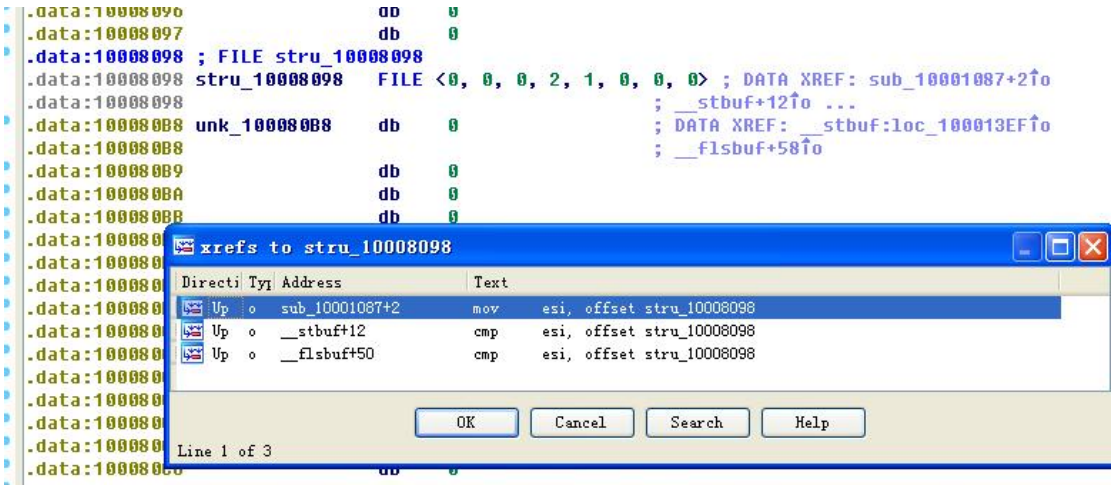
接着调用 getprocaddress 获取 dll3getstructure 的地址,然后 call [ebp+var_10]执行该函数

我们使用 ida 载入 dll3.dll,先看 dll3print

和之前的分析过程一样，我们知道它要打印的数据为 widecharstr,双击跟进后查看交叉引用

可以看到在它之前调用了 Multibyteyowidechar,这个函数的作用就是将多字节的形式转换成宽字符的形式。它要转换的内容是 lpmultibytestr，其实就是上面的 ping…字符串

将其转换成宽字符的形式后将其保存到 widecharstr



Q7 运行或调试 lab09-03.exe 时，会看到打印出三块神秘数据。dll1 的神秘数据，dll2 的神秘数据，dll3 的神秘数据分别是什么

A7.神秘数据 1 是当前进程的标识。神秘数据 2 是打开 temp.xt 文件的句柄，神秘数据 3 是字符串 ping www. malwareanalysisbook. com 在内存中的位置

```
.text:10001087
.text:10001087                  push    ebx
.text:10001088                  push    esi
.text:10001089                  mov     esi, offset stru_10008098
.text:1000108E                  push    edi
.text:1000108F                  push    esi
.text:10001090                  push    1
.text:10001092                  call    __lock_file2
.text:10001097                  push    esi
.text:10001098                  call    __stbuf
.text:1000109D                  mov     edi, eax
.text:1000109F                  lea     eax, [esp+18h+arg_4]
.text:100010A3                  push    eax             ; int
.text:100010A4                  push    [esp+1Ch+arg_0] ; int
.text:100010A8                  push    esi             ; FILE *
.text:100010A9                  call    sub_10001488
.text:100010AE                  push    esi
.text:100010AF                  push    edi
.text:100010B0                  mov     ebx, eax
.text:100010B2                  call    __ftbuf
.text:100010B7                  push    esi
.text:100010B8                  push    1
.text:100010BA                  call    __unlock_file2
```

接着分析 dll3getstructure

可以看到这个函数有一个参数，将其赋给 eax

结合分析 exe 时看到的调用

```
; Exported entry   1. DLL3GetStructure


; Attributes: bp-based frame

public DLL3GetStructure
DLL3GetStructure proc near

arg_0= dword ptr  8

push    ebp
mov     ebp, esp
mov     eax, [ebp+arg_0]
mov     dword ptr [eax], offset dword_1000B0A0
pop     ebp
retn
DLL3GetStructure endp
```

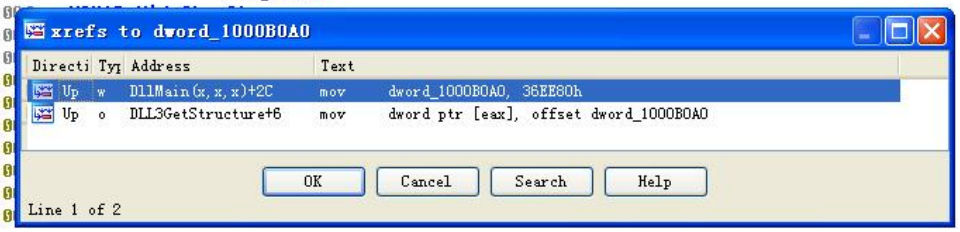
可以看到这个函数有一个参数，将其赋给 eax，结合分析 exe 时看到的调用

而这个 buffer 其实就是一个局部变量，也就是说在调用 dll3strcture 时其参数就是局部变量的地址，在获取到地址之后，把 dword 的值赋给 eax 指向的地址中，也就是前面说的 buffer

查看交叉引用，看第一个 w 的可以看到是一系列的 move 赋值的操作

这些数据的地址会保存在 buffer 里，而 buffer 保存在 ecx，而之后 ecx 是作为 netschedulejobadd 的第二个参数

```
.data:1000B09D                  db    ? ;
.data:1000B09E                  db    ? ;
.data:1000B09F                  db    ? ;
.data:1000B0A0 dword_1000B0A0   dd ?                    ; DATA XREF: DllMain(x,x,x)+2C↑w
.data:1000B0A0                                          ; DLL3GetStructure+6↑o
.data:1000B0A4 dword_1000B0A4   dd ?                    ; DATA XREF: DllMain(x,x,x)+36↑w
.data:1000B0A8 byte_1000B0A8    db ?                    ; DATA XREF: DllMain(x,x,x)+40↑w
.data:1000B0A9 byte_1000B0A9    db ?                    ; DATA XREF: DllMain(x,x,x)+47↑w
.data:1000B0AA                  align 4
.data:1000B0AC dword_1000B0AC   dd ?                    ; DATA XREF: DllMain(x,x,x)+22↑w
.data:1000B0B0                  align 20h
```



这段其实就是 at_info 结构体的数据

Q6.当 lab09-03.exe 调用 NetScheduleJobAdd 创建一个 job 时，从哪里获取第二个参数的数据

A6.Lab09-03.exe 从 DLL3GetStructure 中获取 NetScheduleJobAdd 调用的缓冲区，它动态地解析获得第二个参数的数据

```
; Flags 60000020: Text Executable Readable
; Alignment     : default

; Segment type: Pure code
; Segment permissions: Read/Execute
_text segment para public 'CODE' use32
assume cs:_text
;org 1D1000h
assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing


; Attributes: bp-based frame

; BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
_DllMain@12 proc near

hinstDLL= dword ptr  8
fdwReason= dword ptr  0Ch
lpvReserved= dword ptr  10h

push    ebp
mov     ebp, esp
push    0               ; hTemplateFile
push    80h             ; dwFlagsAndAttributes
push    2               ; dwCreationDisposition
push    0               ; lpSecurityAttributes
push    0               ; dwShareMode
push    40000000h       ; dwDesiredAccess
push    offset FileName ; "temp.txt"
call    ds:CreateFileA
mov     dword_1DB078, eax
mov     al, 1
pop     ebp
retn    0Ch
_DllMain@12 endp
```

解决最后一个问题，ida 载入 dll2.dll 时选择 manual load，这里的地址是 1d1000

是因为这是 text 段，这和 od 中看到的是一样的，说明 ida 与 od 加载的地址匹配了

Q8 如何将 dll2.dll 加载到 IDA 中，使得它与 ollydbg 使用的加载地址匹配

A8.当使用 IDAPro 加载 DLL 时选择手动加载，当提示时，输入新的映像基准地址。本例中，地址是 0x1d0000。

Yara 规则的编写与检测

```
UrlRequest C:\Users\53653_000\Desktop\duku\PracticalMalwareAnalysis-Labs\Practic
al Malware Analysis Labs\BinaryCollection\Chapter_9L\Lab09-02.exe
cmd C:\Users\53653_000\Desktop\duku\PracticalMalwareAnalysis-Labs\Practical Malw
are Analysis Labs\BinaryCollection\Chapter_9L\Lab09-02.exe
SOCKET C:\Users\53653_000\Desktop\duku\PracticalMalwareAnalysis-Labs\Practical M
alware Analysis Labs\BinaryCollection\Chapter_9L\Lab09-02.exe
UrlRequest C:\Users\53653_000\Desktop\duku\PracticalMalwareAnalysis-Labs\Practic
al Malware Analysis Labs\BinaryCollection\Chapter_9L\DLL1.dll
DLL C:\Users\53653_000\Desktop\duku\PracticalMalwareAnalysis-Labs\Practical Malw
are Analysis Labs\BinaryCollection\Chapter_9L\DLL1.dll
UrlRequest C:\Users\53653_000\Desktop\duku\PracticalMalwareAnalysis-Labs\Practic
al Malware Analysis Labs\BinaryCollection\Chapter_9L\Lab09-03.exe
DLL C:\Users\53653_000\Desktop\duku\PracticalMalwareAnalysis-Labs\Practical Malw
are Analysis Labs\BinaryCollection\Chapter_9L\Lab09-03.exe
UrlRequest C:\Users\53653_000\Desktop\duku\PracticalMalwareAnalysis-Labs\Practic
al Malware Analysis Labs\BinaryCollection\Chapter_9L\DLL2.dll
DLL C:\Users\53653_000\Desktop\duku\PracticalMalwareAnalysis-Labs\Practical Malw
are Analysis Labs\BinaryCollection\Chapter_9L\DLL2.dll
UrlRequest C:\Users\53653_000\Desktop\duku\PracticalMalwareAnalysis-Labs\Practic
al Malware Analysis Labs\BinaryCollection\Chapter_9L\Lab09-01.exe
cmd C:\Users\53653_000\Desktop\duku\PracticalMalwareAnalysis-Labs\Practical Malw
are Analysis Labs\BinaryCollection\Chapter_9L\Lab09-01.exe
EXE C:\Users\53653_000\Desktop\duku\PracticalMalwareAnalysis-Labs\Practical Malw
are Analysis Labs\BinaryCollection\Chapter_9L\Lab09-01.exe
Regedit C:\Users\53653_000\Desktop\duku\PracticalMalwareAnalysis-Labs\Practical
Malware Analysis Labs\BinaryCollection\Chapter_9L\Lab09-01.exe
UrlRequest C:\Users\53653_000\Desktop\duku\PracticalMalwareAnalysis-Labs\Practic
al Malware Analysis Labs\BinaryCollection\Chapter_9L\DLL3.dll
DLL C:\Users\53653_000\Desktop\duku\PracticalMalwareAnalysis-Labs\Practical Malw
are Analysis Labs\BinaryCollection\Chapter_9L\DLL3.dll
```

根据我们之前的字符串分析，编写以下的 yara 规则：

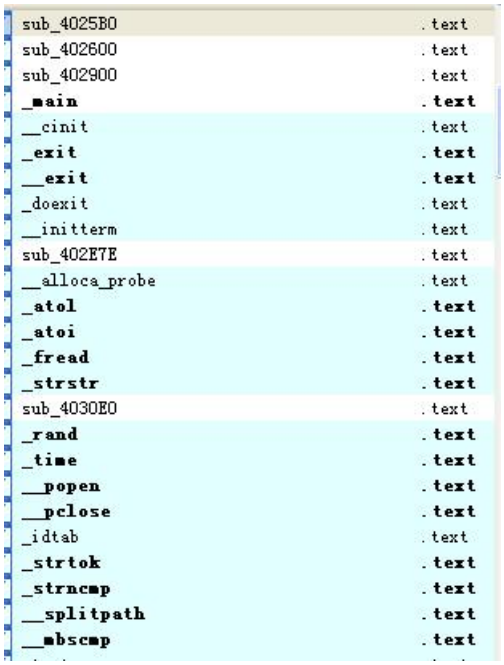
```
import "pe"
rule UrlRequest {
        strings:
                $http = "http"
                $GET = "GET" nocase
                $com = /[a-zA-Z0-9_]*.com/
        condition:
                $http or $GET or $com
}
rule cmd {
        strings:
                $name = "cmd" nocase
        condition:
                $name
}
rule EXE {
        strings:
                $exe = /[a-zA-Z0-9_]*.exe/
        condition:
                $exe
}
rule Regedit{
        strings:
                $system = "system32"
                $software = "SOFTWARE"
        condition:
                $system or $software
```

```
}
rule DLL {
        strings:
                $dll = "DLL"
        condition:
                $dll
}
rule SOCKET {
        strings:
                $name = "Socket"
        condition:
                $name
}
```

Ida python 的编写

通过观察三个 lab 里面的函数名，我们可以根据相应的函数编写 ida python 脚本。

运行脚本即可搜索函数当中的字符串，得到目标结果。



Lab09-01.exe

```
#coding:utf-8 from idaapi import *
```

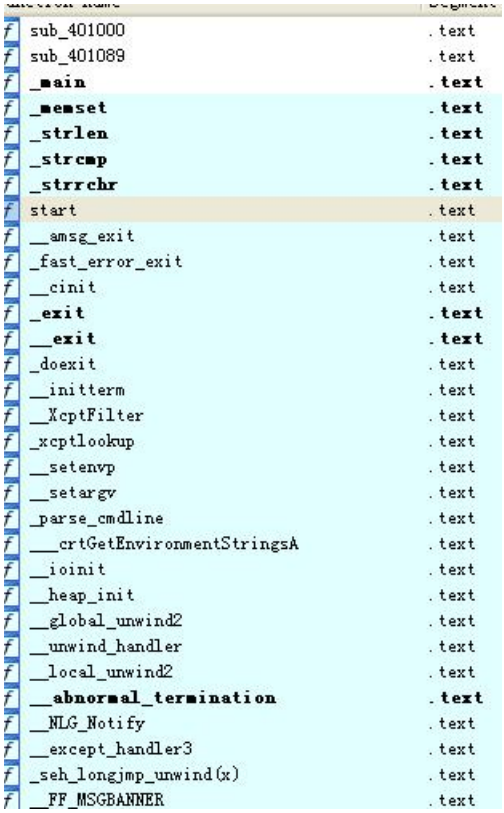
设置颜色

```
def judgeAduit(addr): ''' not safe function handler ''' MakeComm(addr,"### AUDIT HERE ###") SetColor(addr,CIC_ITEM,0x0000ff)
#set backgroud to red pass
```

函数标识

```
def flagCalls(danger_funcs): ''' not safe function finder ''' count = 0 for func in danger_funcs:
faddr = LocByName( func )
if faddr != BADADDR: # Grab the cross-references to this address
cross_refs = CodeRefsTo( faddr, 0 )
for addr in cross_refs: count += 1 Message("%s[%d] calls 0x%08x\n"%(func,count,addr))
judgeAduit(addr)
```

if **name** == '**main**': ''' handle all not safe functions ''' print "------------------------------" # 列表存储需要识别的函数
danger_funcs = ["___crtGetEnvironmentStringsA","__heap_init","__abnormal_termination","_free","RtlUnwind"]
flagCalls(danger_funcs)



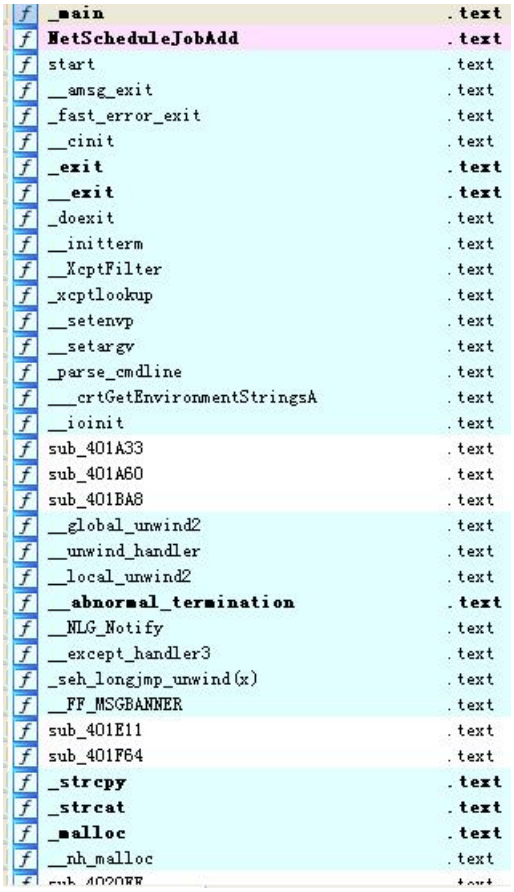
Lab09-02.exe

```
#coding:utf-8
from idaapi import *

# 设置颜色
def judgeAduit(addr):
    '''
    not safe function handler
    '''
    MakeComm(addr,"### AUDIT HERE ###")
    SetColor(addr,CIC_ITEM,0x0000ff)    #set backgroud to red
    pass

# 函数标识
def flagCalls(danger_funcs):
    '''
    not safe function finder
    '''
    count = 0
    for func in danger_funcs:
        faddr = LocByName( func )
        if faddr != BADADDR:
            # Grab the cross-references to this address
            cross_refs = CodeRefsTo( faddr, 0 )
            for addr in cross_refs:
                count += 1
                Message("%s[%d] calls 0x%08x\n"%(func,count,addr))
```

```
                judgeAduit(addr)

if __name__ == '__main__':
    '''
    handle all not safe functions
    '''
    print "--------------------------------"
    # 列表存储需要识别的函数
    danger_funcs = ["_XcptFilter","_initterm","nullsub_1","_except_handler3","_controlfp"]
flagCalls(danger_funcs)
```

Lab09-03.exe

```
#coding:utf-8 from idaapi import *
设置颜色
def judgeAduit(addr): ''' not safe function handler ''' MakeComm(addr,"### AUDIT HERE ###") SetColor(addr,CIC_ITEM,0x0000ff)
#set backgroud to red pass
函数标识
def flagCalls(danger_funcs): ''' not safe function finder ''' count = 0 for func in danger_funcs:
faddr = LocByName( func )
if faddr != BADADDR: # Grab the cross-references to this address
cross_refs = CodeRefsTo( faddr, 0 )
for addr in cross_refs: count += 1 Message("%s[%d] calls 0x%08x\n"%(func,count,addr))
judgeAduit(addr)
if name == 'main': ''' handle all not safe functions ''' print "--------------------------------" # 列表存储需要识别的函数
danger_funcs = ["start","_XcptFilter","_initterm","__setdefaultprecision","_controlfp"] flagCalls(danger_funcs)
```



四、实验结论及心得体会

这一次的实验是恶意代码与防治分析的 Lab9 实验，对理论课上讲的 IDA Python 编写技术有了一定的了解，也对 IDA Pro 的使用比如说交叉引用、语句跳转、反汇编分析等更加的熟练。在本次实验中，也对所检测程序编写了相应的 yara 规则，对于 yara 规则的编写也更加的熟练。本次实验当中，我重新回顾了之前学习过的 ollydbg 的使用，通过使用 ollydbg 了解到了一些关于栈寄存器等等目标的信息，在栈寄存器当中我们不断的跟进程序流程，对 ida 与 ollydbg 协同运用，解决相应的目标问题有了新的理解。通过本次实验，也知道了一些应用程序的代码结构和其基本功能，认识到自己作为一名信息安全专业学生的责任，需要我们用更加严谨认真的态度学习新的知识和思路。

未来我还将联系操作系统和本课程当中关于进程和线程的相关知识，对 windows 恶意代码的特殊作用进行进一步的讨论。