

**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

1. **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。

1. **实验原理**
   1. 实验环境

Windows xp，VMWARE，Windows11 ,win8.1

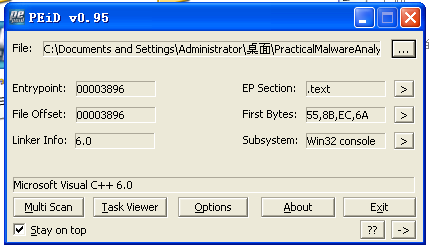
* 1. 实验工具

Ida pro 6.6 ida python,yara

1. **实验过程**

Lab9-1分析

1、首先查壳；



无壳；

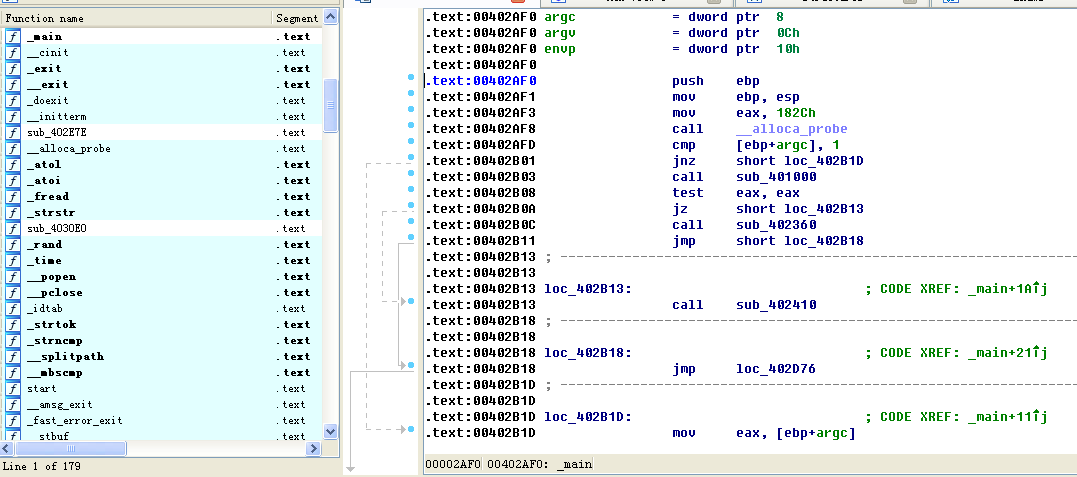
1. 拖进IDA Pro查看导入函数；

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；

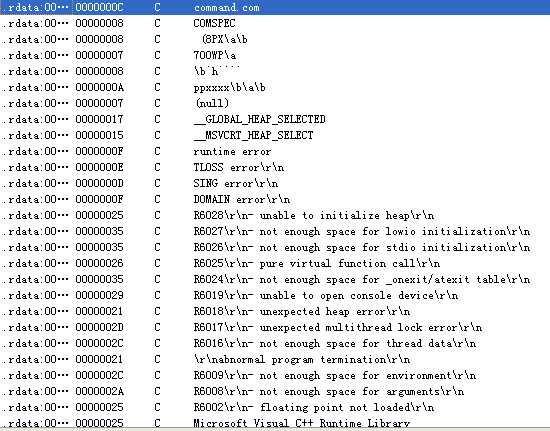
.com、.bat和.cmd这种有趣的文件后缀；

http/1.0\r\n\r\n和GET这种网络关键字；

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

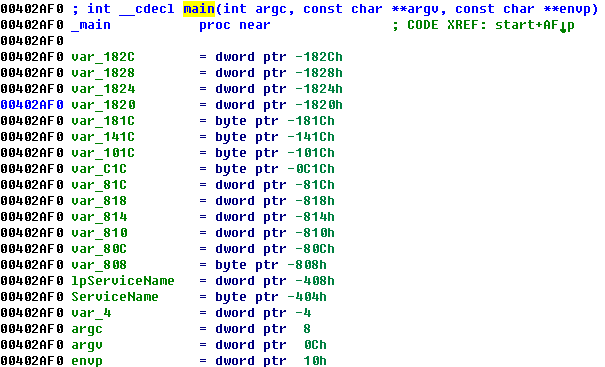
%SYSTEMOOT%\\SYSTEM32\\敏感路径；

更加确认了有网络行为，并且有远程指令的嫌疑；



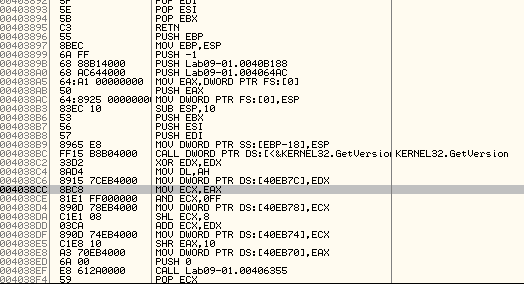
4、分析行为；

使用IDA Pro查看main()入口，再使用OllyDbg跟踪；



#### ****问题1.如何让这个恶意代码安装自身？****

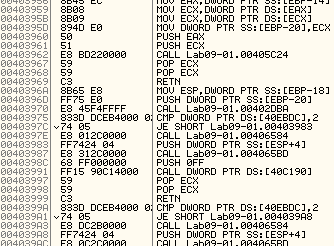
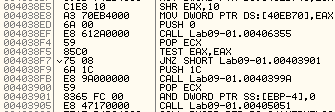
我们先打开OllyDbg，我们可以看到我们停到了这里这个地方，这里对我们来说并不是什么重要的东西，我们可以按F8(step-over)往后慢慢一步一步走，其实这时候按F7（step-into)是最保险的，因为你不知道什么时候会调用到函数，然后如果你想回退到开始执行的地方，可以按Ctrl+F2来回溯到函数一开始执行的地方



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

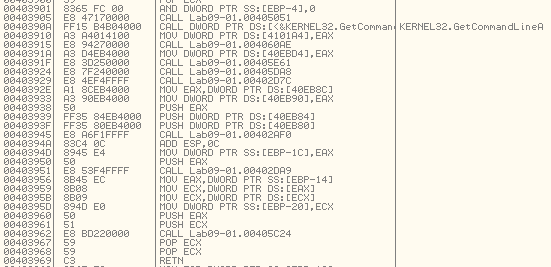


到现在这些函数都是程序执行前的初始化，这里是获取导入库的地址的

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。

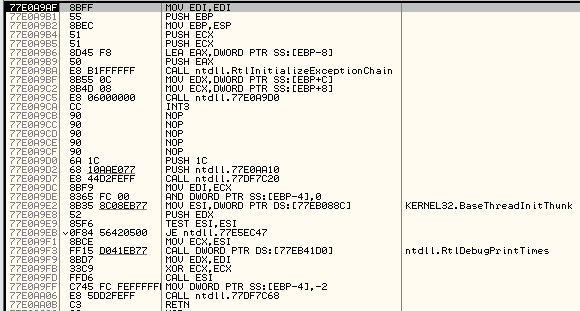
然后我们来到这个CALL的地方，这个也是初始化的东西，但是我们注意到下面这个CALL的函数是GetCommandLineA，这个函数是获得用户输入的函数，说明这个程序需要用户提供一个参数。

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。

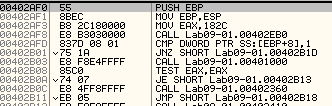
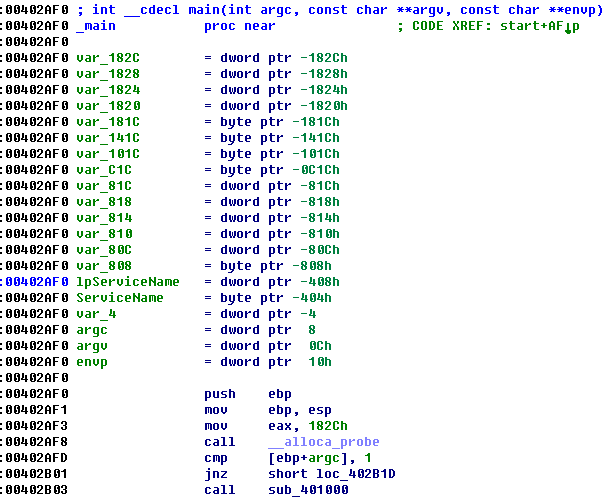


下面就是这三个函数的CALL，我们进去第一个调用405E61处，然后我们会发现这个函数其实就是调用模块的。第二个调用也是用于获取文件名字的。第三个调用获得了当前进程的ID，这其实也是程序执行过程中，初始化的过程。

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



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。



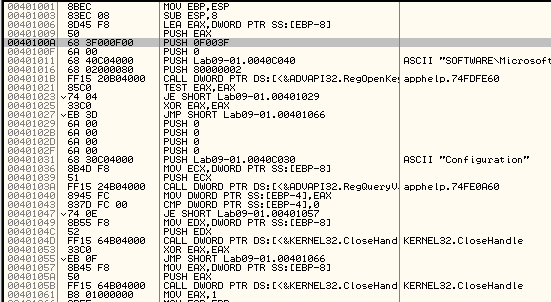
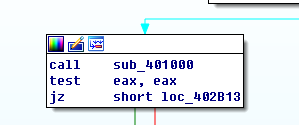
然后这时到了程序转移到被调用函数中执行，这个期间，地址ebp+0指向的是外面那个函数的ebp指针，ebp+4指向的是RET返回地址，ebp+8就是我们的第一个入参，也就是这里的Arg1。

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。

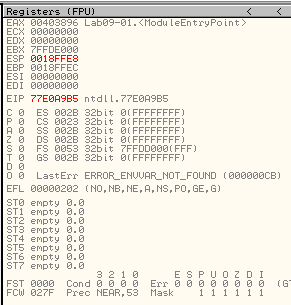
而是继续往下执行，然后从00402B03开始的话，就到了IDA的这里



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

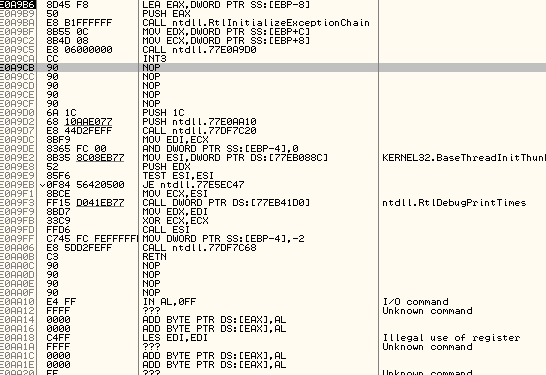


函数的返回值在EAX中存储，这时候是2

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

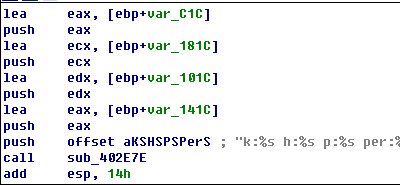
就是检测系统中是否存在那个注册表键，如果没有的话，就退出这个函数，并返回0，那如果成功呢，我们这里手动修改了一下eax看看

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1

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。

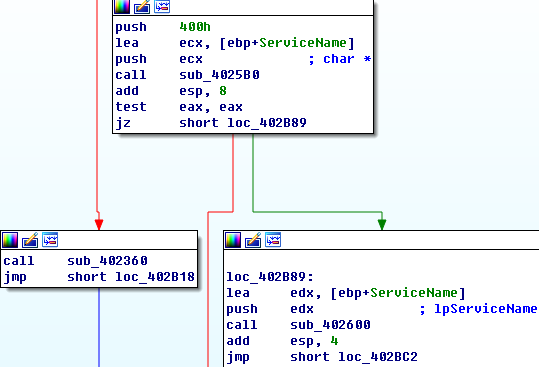
其实也就是IDA中的sub\_402410。这里用ShellExecuteA打开了cmd.exe，这应该是这个恶意程序的一种伪造，如果用户没有输入任何的参数，或者没有发现那个注册表项，再或者查询注册表失败，就来执行这个ShellExecuteA通过cmd.exe来运行一些命令。

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

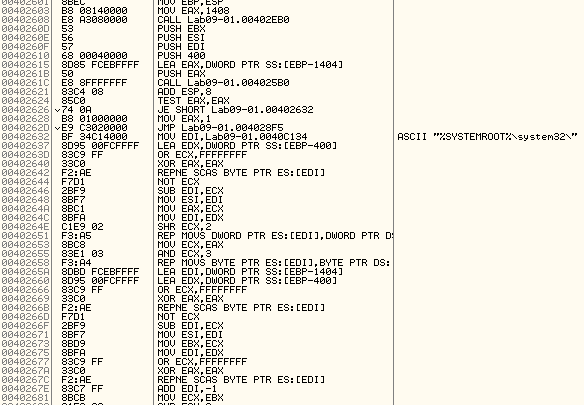
然后我们从函数中返回之后就是来到了这里，这有个test，我们成功了，所以返回值是0，然后test之后ZF=0，JE跳转

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

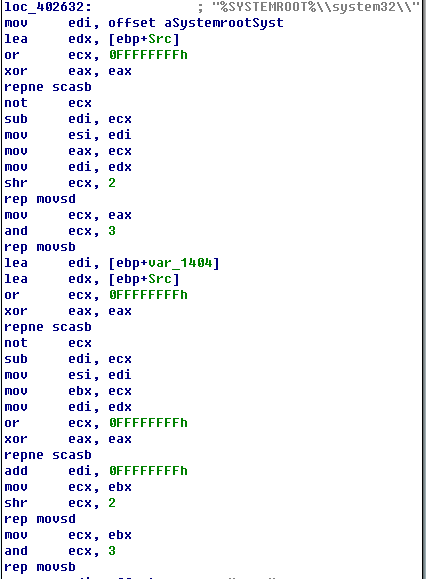
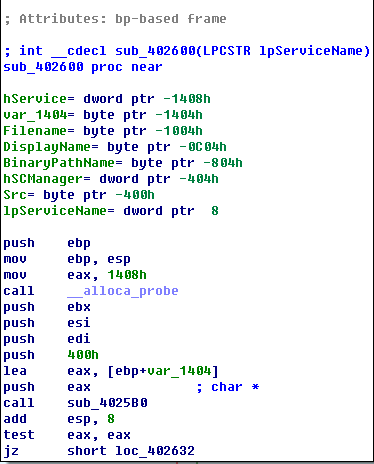


我们可以大概看出来，这个EDX其实就是刚刚上面那个函数处理后的返回结果Lab09-01

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



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)

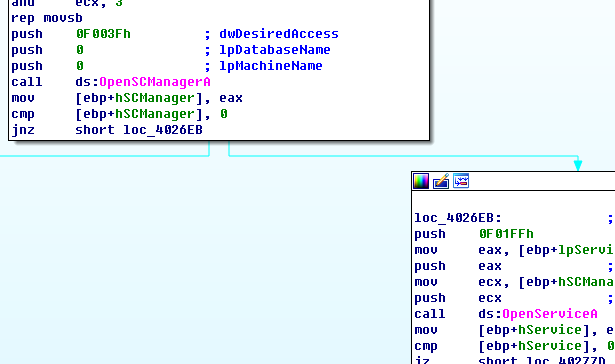
这里函数就有这么几个入参，执行看看，然后这时候的EAX返回值就是0了

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

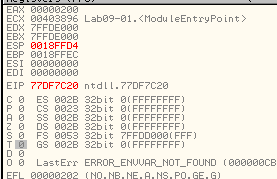
这里显示是的计算[EDI]的字符长度也就是上面那个字符串的长度

这么一串指令到最后的



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。

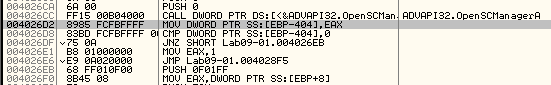
前两个参数为0说明这个服务控制器是在本地上，这时候的寄存器。



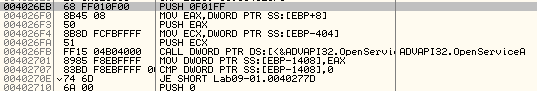
可以看出上面那个操作就是把这个字符串拼接成EDX所示的样子

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



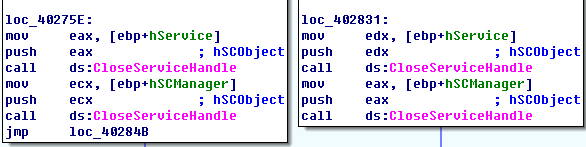
之后去到004026EB。



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

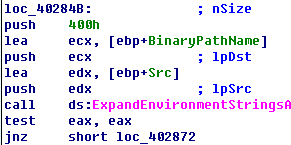
这里调用成功之后，返回值肯定不为0，所以和0执行cmp之后，ZF=0，之后JNZ肯定就会跳转了



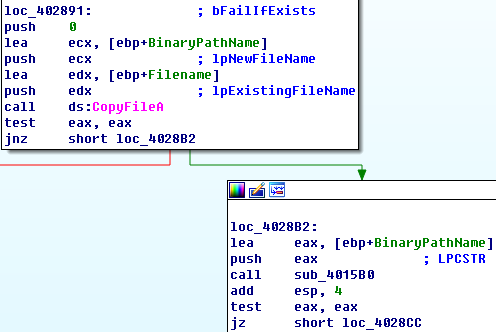
跳转之后的绿线就是执行一写关闭操作，来关闭打开的Handle们

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

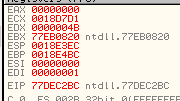


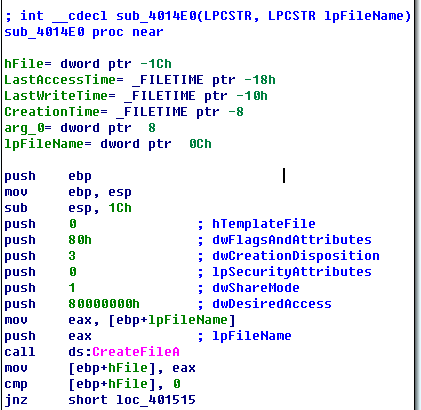
然后执行成功之后继续往下，这里两个入参，edx和ecx，lpNewFileName的值是ecx，然后lpExistingFileName也就是edx也是上面我们刚刚的返回值，这里要将这个可执行文件赋值到system32下面。执行成功之后就会跳到这里



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。

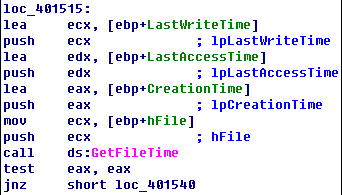




我们还是按照先看入参然后看返回值的做法看看，这个函数的入参是eax，这个的值是

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

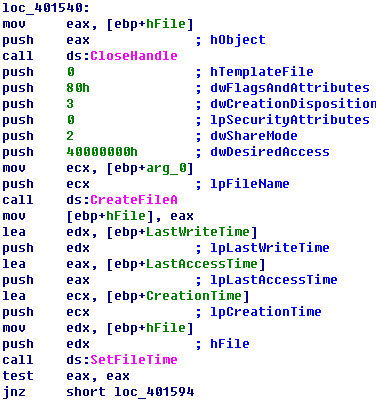
这里我们执行后会发现返回值不是0，然后就是和0就行cmp，ZF=0，则JNZ跳转，走绿线，也就是这里



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。

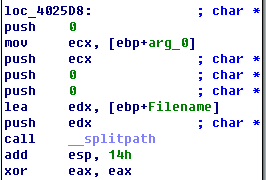
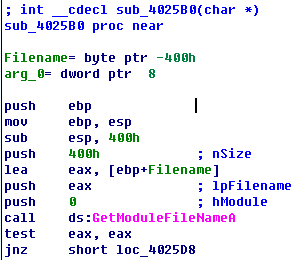


然后设置这个文件的时间和kernel32.dll的一样

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



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**

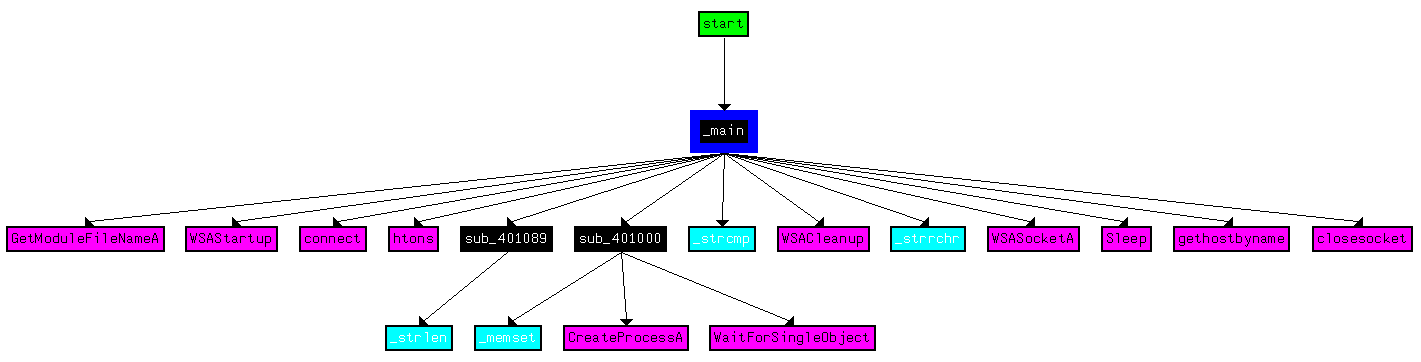
**问题一：字符串由strings得到**



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。

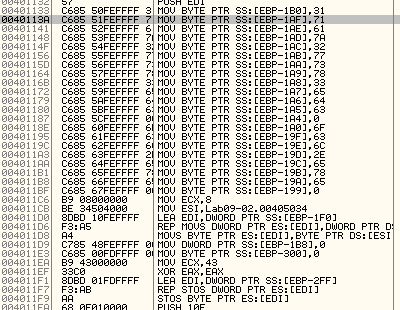


问题3：

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：

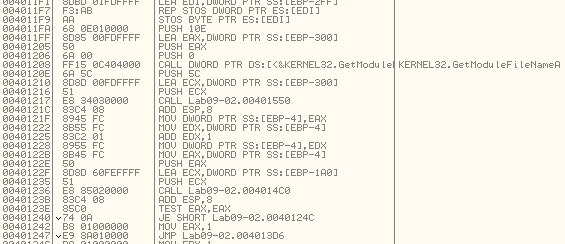
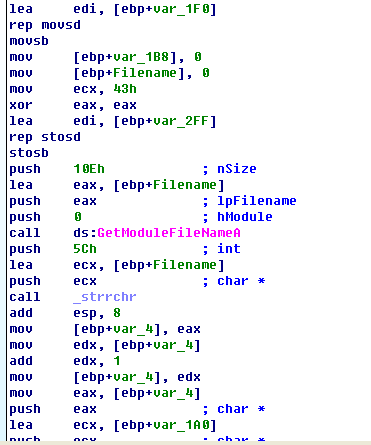
首先看到了一大串的mov指令



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。

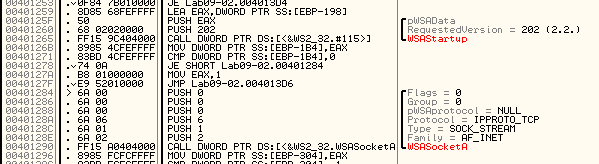
第一串并不能看出什么，第二串字符有可能是一个文件名。继续向下分析：

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。

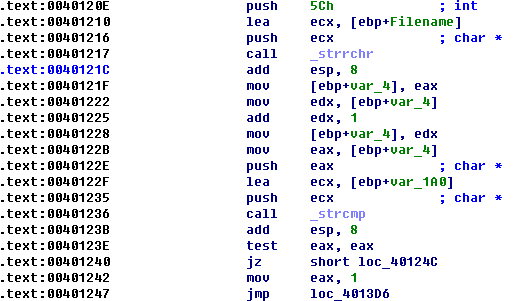


继续向下走：

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。



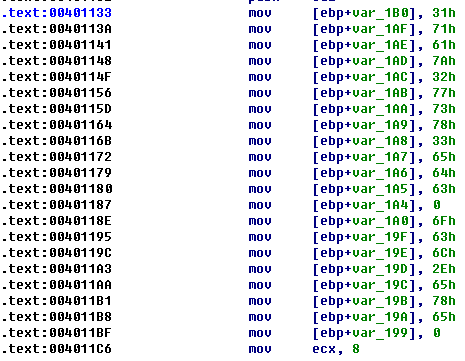
这里看到调用了004014C0，OD没有解析出它的名称，但是我们看到了两个参数，一个是前面所怀疑的ocl.exe，另一个就是文件名了。



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。

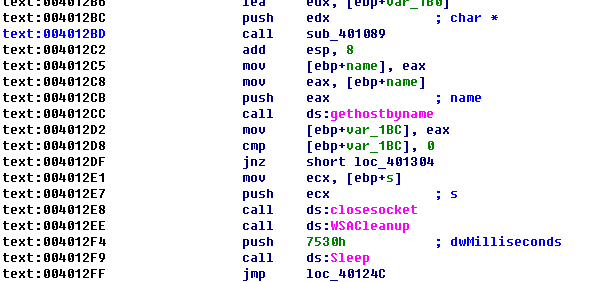
问题4：快捷键G跳转到地址401133：

然后发现，这个就是一系列的赋值操作。

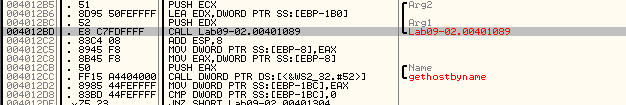


问题5：

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：



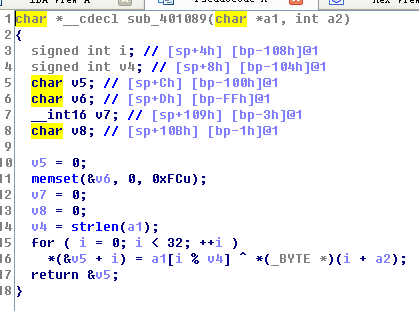
发现了第一个参数就是之前的第一个字符串，第二个参数是一个数据缓冲区，暂时无法识别。

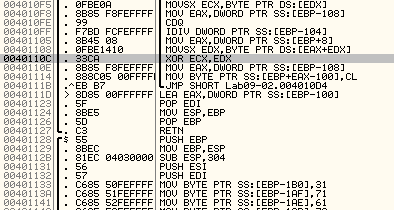


问题6:

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





这里可以看到是由edx和ecx进行异或操作，F9运行：

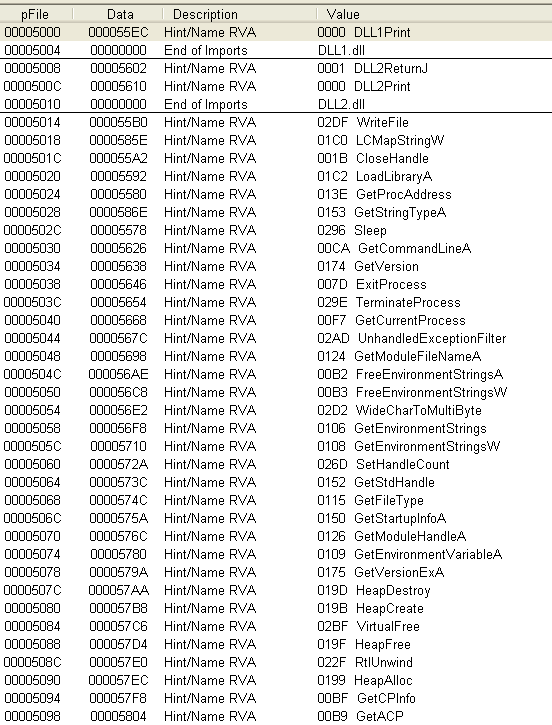
这样就看到了异或操作之后出来的一个网址，这个就是恶意代码使用的域名。

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



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**



问题1：

首先用peview查看导入表：

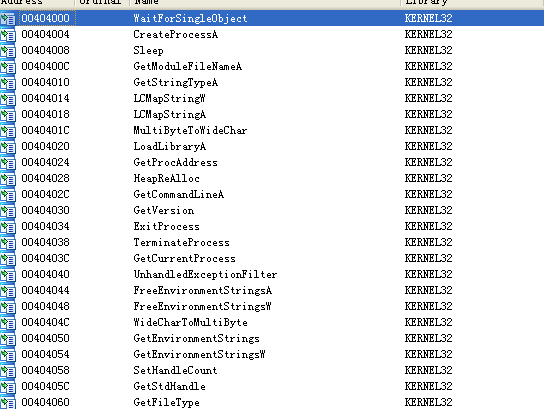
通过peview可以看到，分别导入了以下四个dll。

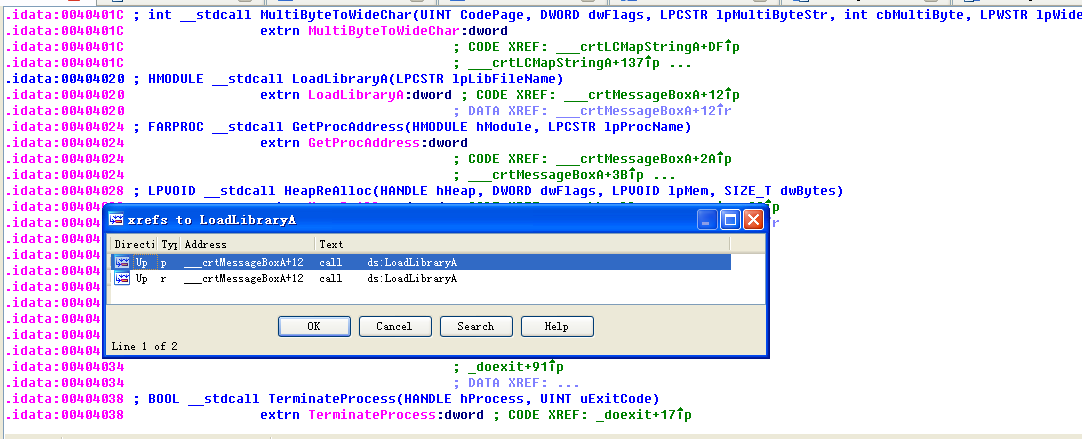
还有些dll可能在程序运行时加载，需要ida分析，一般会需要loadlibrary

Q1.lab09-03.exe导入了哪些dll

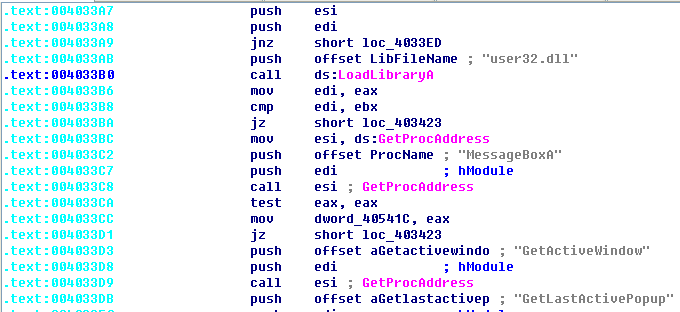
A1.导入的dll有：kernel32.dll、NetAPI32.dll、DLLI.dll和DLL2.dl

在imports窗口找到loadlibrary

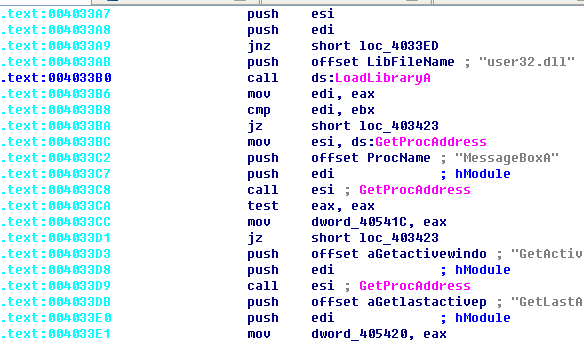




p表示函数调用，r表示读取，我们要关注的调用，先看第一个p，可以看到

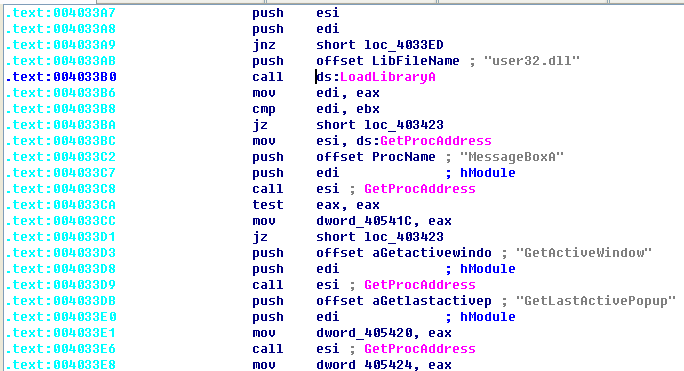


在调用时其第一个参数是dll3.dll，再看一下第二个



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



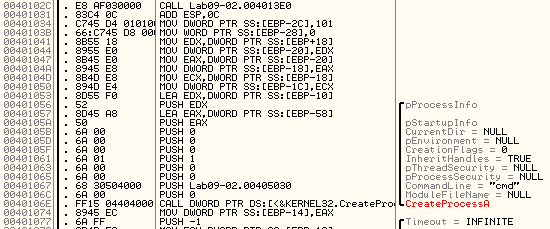
dll2，dll3都是如此

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

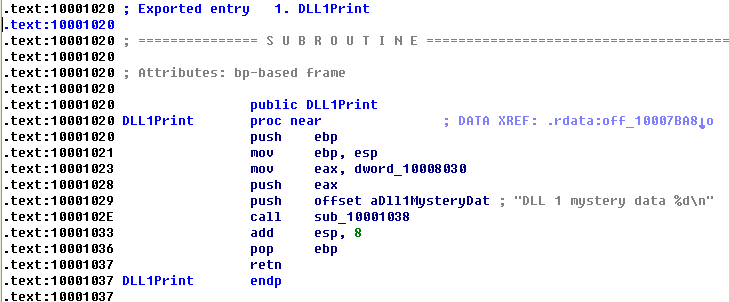


此时就可以看到dll3的地址是001f0000

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。

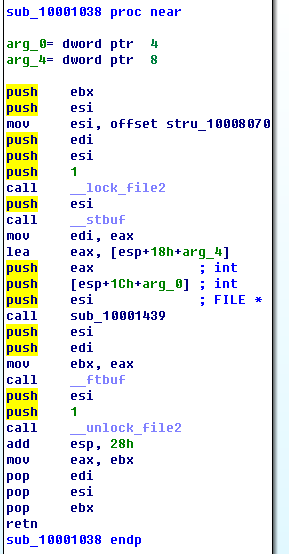
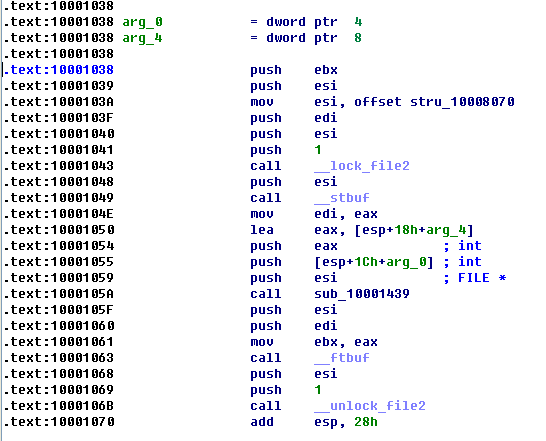
为了回答第4个问题:ida载入dll1.dll,找到dll1print



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

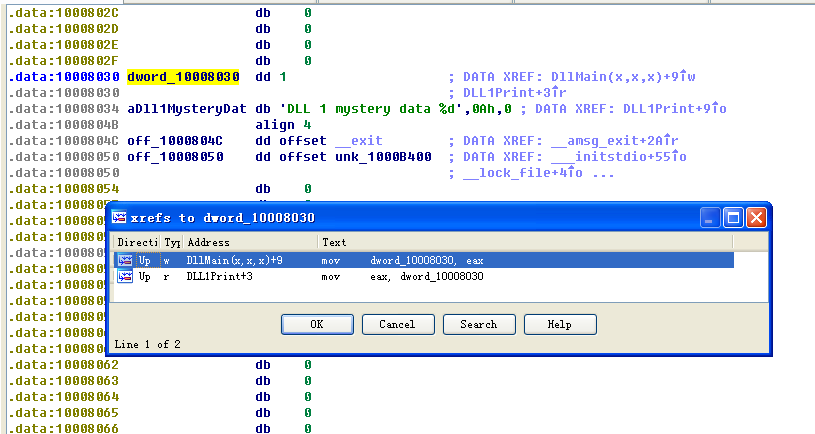
调用了sub\_10001038,跟进去



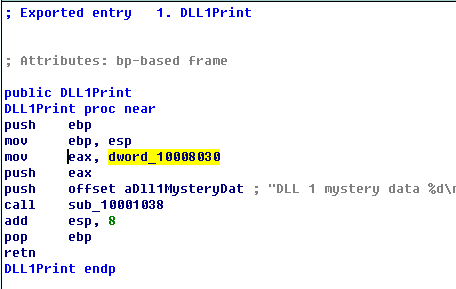
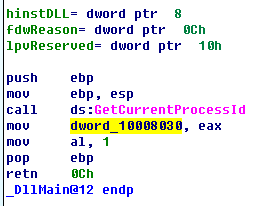
可以看到这个函数是有两个参数,第一个参数是字符串,字符串里有%d,\n。

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。



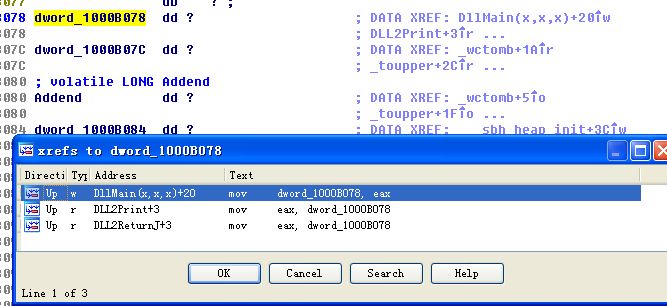
可以看到它保存的其实就是GetCurrentProcessId的返回值，也就是当前进程的id值。找到了writefile，看到第一个参数是hfile,保存的是要写入的文件句柄，而第二个参数是要写入的内容，也就是malware….字符串，为了知道文件名称，所以需要分析的是第一个参数，可以看到它其实是dll2returnj的返回值。所以需要ida分析dll2.dll，先看一下dll2print

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

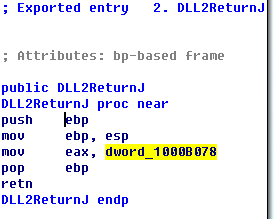
分析w



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



主要就是讲dword也就是temp.txt的句柄赋给eax，之后就返回了

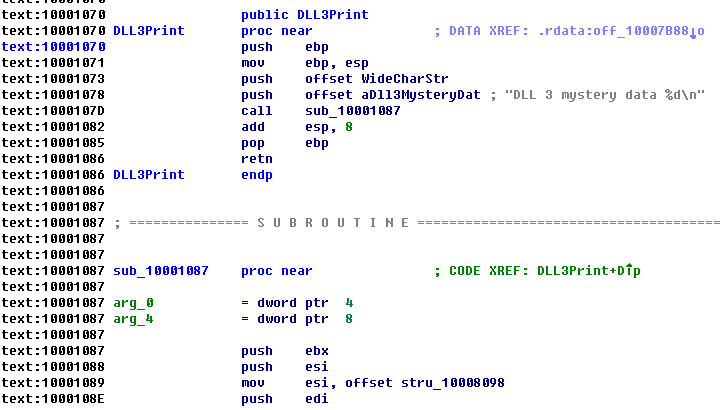
可以看到，dll2reurnj返回的eax就是temp.txt文件的句柄

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



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

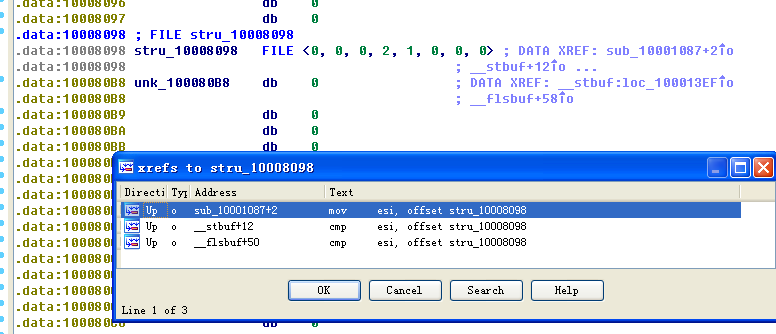
接着调用getprocaddress获取dll3getstructure的地址,然后call [ebp+var\_10]执行该函数

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

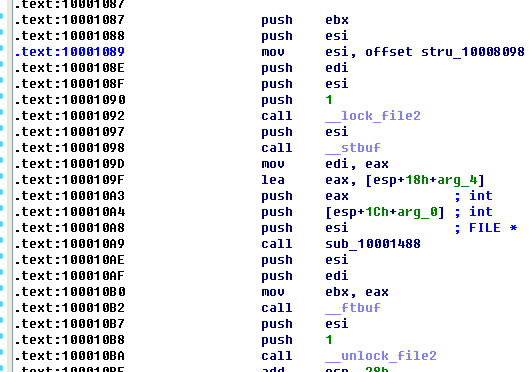
可以看到在它之前调用了Multibyteyowidechar,这个函数的作用就是将多字节的形式转换成宽字符的形式。它要转换的内容是lpmultibytestr，其实就是上面的ping…字符串

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

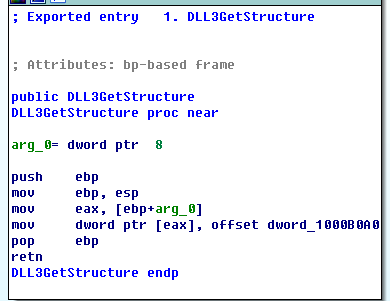
A7.神秘数据1是当前进程的标识。神秘数据2是打开temp.xt文件的句柄，神秘数据3是字符串ping www. malwareanalysisbook. com在内存中的位置



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

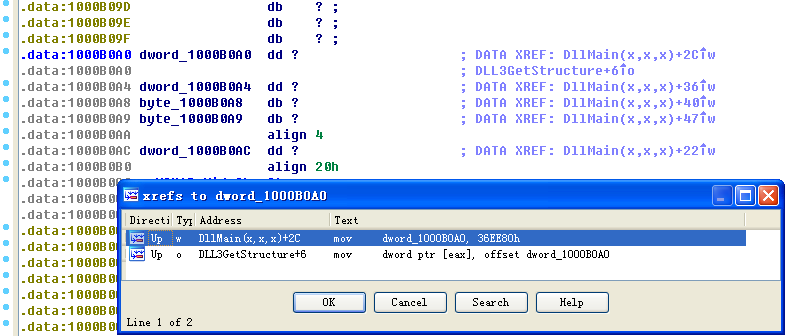


可以看到这个函数有一个参数，将其赋给eax，结合分析exe时看到的调用

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

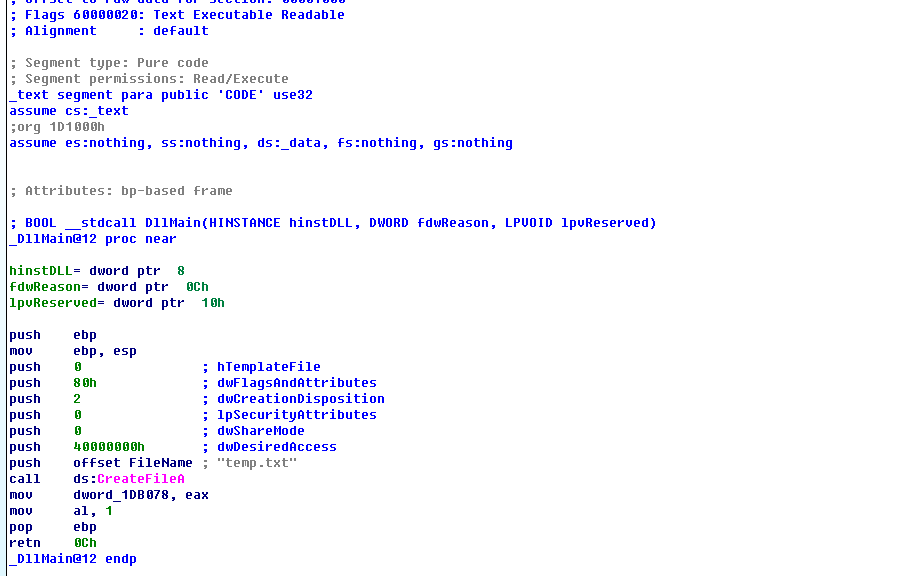
这些数据的地址会保存在buffer里，而buffer保存在ecx，而之后ecx是作为netschedulejobadd的第二个参数



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



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**



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:

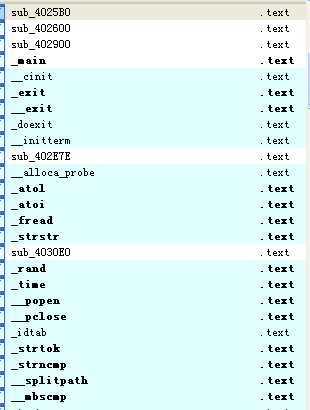
$name

}

**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 \*

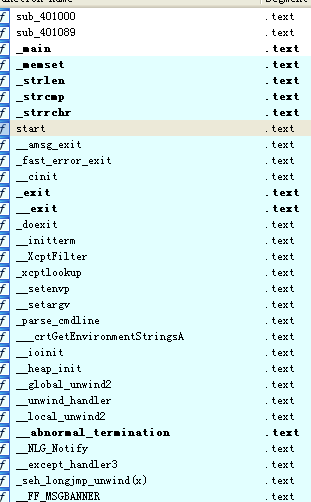
设置颜色

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 \*

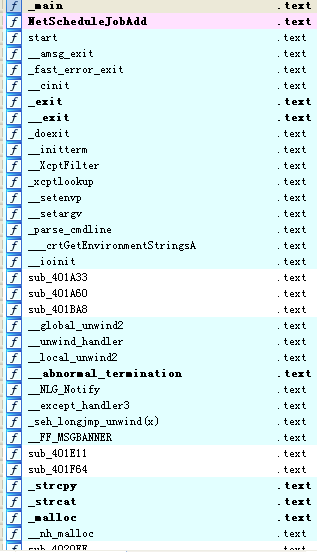
设置颜色

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)



1. **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。