

个人信息

学号: 1911410

姓名: 付文轩

专业: 信息安全

lab 13-01

问题1

比较恶意代码中的字符串（字符串命令的输出）与动态分析提供的有用信息，基于这些比较，哪些元素可能被加密？

首先使用strings简单查看一下这个程序

```
GetProcAddress
LoadLibraryA
SetStdHandle
FlushFileBuffers
MultiByteToWideChar
LCMapStringA
LCMapStringW
GetStringTypeA
GetStringTypeW
CloseHandle
7"e
Mozilla/4.0
http://%s/%s/
Could not load exe.
Could not locate dialog box.
Could not load dialog box.
Could not lock dialog box.
%02x
`ne
`ne
lse
<se
TRE
<<<<< H
`y?
e^c
[
Q^ _j2
LLL
KIZKORXZWUZWLZI^ZUZWBHRH
XTU
^]I
^
ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789+/-
EEE
<0PV
```

可以其中有一部分的字符串枚举了所有的大小写字母和数字，以及+/, 很明显这是一个base64编码的内容那么猜测这个恶意代码会存在有base64加密的内容。之后可以看见有一部分看似是乱码的内容：

KIZXORXZWVZLZI^ZUZWBRH, 结合刚刚猜测的会存在有base64编码，想来这里应该就是加密过后的内容。

同时我们还可以发现里面有 http:// 的字样，而且这里是一个格式化字符串，上面还有一个 Mizilla/4.0，暂时还不知道是什么作用。

之后我们实际运行一下这个程序，由于之前分析出来可能会有网络行为，所以这里使用wireshark抓包分析一下

No.	Time	Source	Destination	Protocol	Length	Info
17	6.47540800	192.0.78.25	192.168.159.131	HTTP	461	HTTP/1.1 301 Moved Permanently (text/html)
18	6.58760800	192.0.78.25	192.168.159.131	HTTP	461	HTTP/1.1 301 Moved Permanently (text/html)

Request Version: HTTP/1.1
Status Code: 301
Response Phrase: Moved Permanently
Server: nginx
Date: Sat, 18 Dec 2021 12:40:39 GMT
Content-Type: text/html
Content-Length: 162
Connection: keep-alive
Location: https://www.practicalmalwareanalysis.com/bmlydmFUYs04MMw/
X-ac: 3.mrt_bor

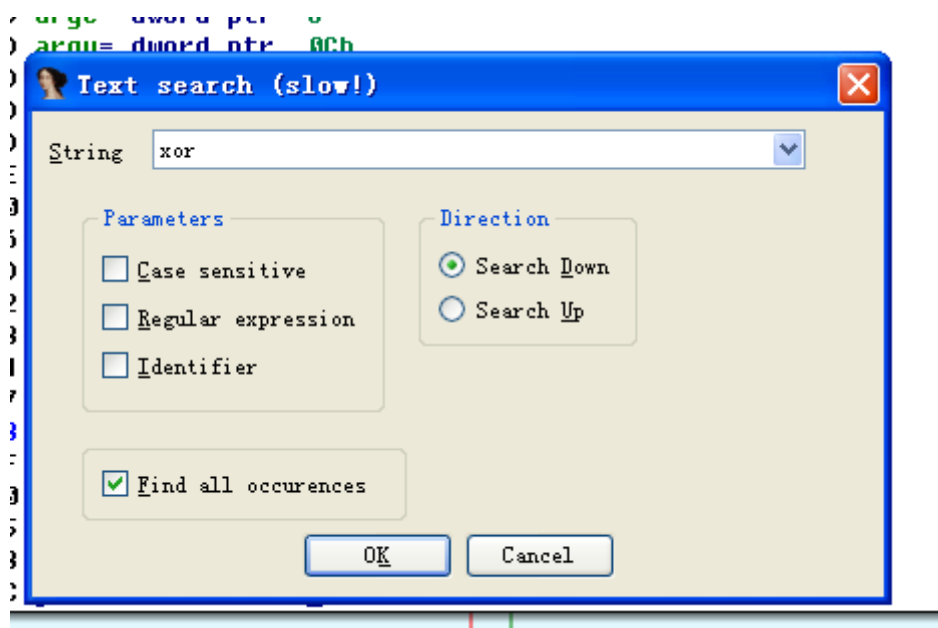
0060	69 6e 78 0d 0a 44 61 74 65 3a 20 53 61 74 2c 20	inx..bat e: Sat,
0070	31 38 20 44 65 63 20 32 30 32 31 20 31 32 3a 34	18 Dec 2 021 12:4
0080	30 3a 33 39 20 47 4d 54 0d 0a 43 6f 6e 74 65 6e	0:39 GMT ..Conten
0090	74 2d 54 79 70 65 3a 20 74 65 78 74 2f 68 74 6d	t-type: text/htm
00a0	6c 0d 0a 43 6f 6e 74 65 6e 74 2d 4c 65 6e 67 74	l..Conte nt-Lengt
00b0	68 3a 20 31 36 32 0d 0a 43 6f 6e 6e 65 63 74 69	h: 162.. Connecti
00c0	6f 6e 3a 20 6b 65 65 70 2d 61 6c 69 76 65 0d 0a	on: keep-alive
00d0	4c 6f 68 61 72 69 61 6e 3a 20 68 74 74 70 79 98	Location: https:
00e0	2f 2f 77 77 72 2e 70 72 61 63 74 69 63 61 6c 6e	//www.pr acticalm
00f0	61 6c 77 61 72 65 61 6e 61 6c 79 73 69 73 2e 63	alwarean alysis.c
0100	6f 6d 2f 62 6d 6c 79 64 6d 46 75 59 53 30 34 46	om/bmlyd mFUYs04M
0110	6d 4d 72 2f 0d 0a 58 2d 61 63 3a 20 33 2e 6e 72	mw/..X- ac: 3.mrt
0120	74 20 5f 62 75 22 20 0d 0a 0d 0a 3c 68 74 6d 6c	_bor .html
0130	3e 0d 0a 3c 68 65 61 64 3e 3c 74 69 74 6c 65 3e	>..<head><title>
0140	33 30 31 20 4d 6f 76 65 64 20 50 65 72 6d 61 6e	301 Move d Perman
0150	65 6e 74 6c 79 3c 2f 74 69 74 6c 65 3e 3c 2f 68	ently</t itle></h
0160	65 61 64 3e 0d 0a 3c 62 6f 64 79 3e 0d 0a 3c 63	ead>..<b od>..<c
0170	65 6e 74 65 72 3e 3c 68 31 3e 33 30 31 20 4d 6f	enter>h 1>301 Mo
0180	76 65 64 20 50 65 72 6d 61 6e 65 6e 74 6c 79 3c	ved Perm anently<
0190	2f 68 31 3e 3c 2f 63 65 6e 74 65 72 3e 0d 0a 3c	/hd></ce nter>..<
01a0	68 72 3e 3c 63 65 6e 74 65 72 3e 6e 67 69 6e 78	hr><cent er>ngin
01b0	3c 2f 63 65 6e 74 65 72 3e 0d 0a 3c 2f 62 6f 64	</center>..</bod
01c0	79 3e 0d 0a 3c 2f 68 74 6d 6c 3e 0d 0a	y>..</ht ml>..

可以发现有一个HTTP请求，并且这个请求是一个GET请求，请求的网址我们之前在string中并没有看见，想来可能和加密的内容有关

问题2

使用IDA搜索恶意代码中字符串“xor”，以此来查找潜在的加密，你发现了哪些加密的类型？

使用IDA打开程序，并进行搜索



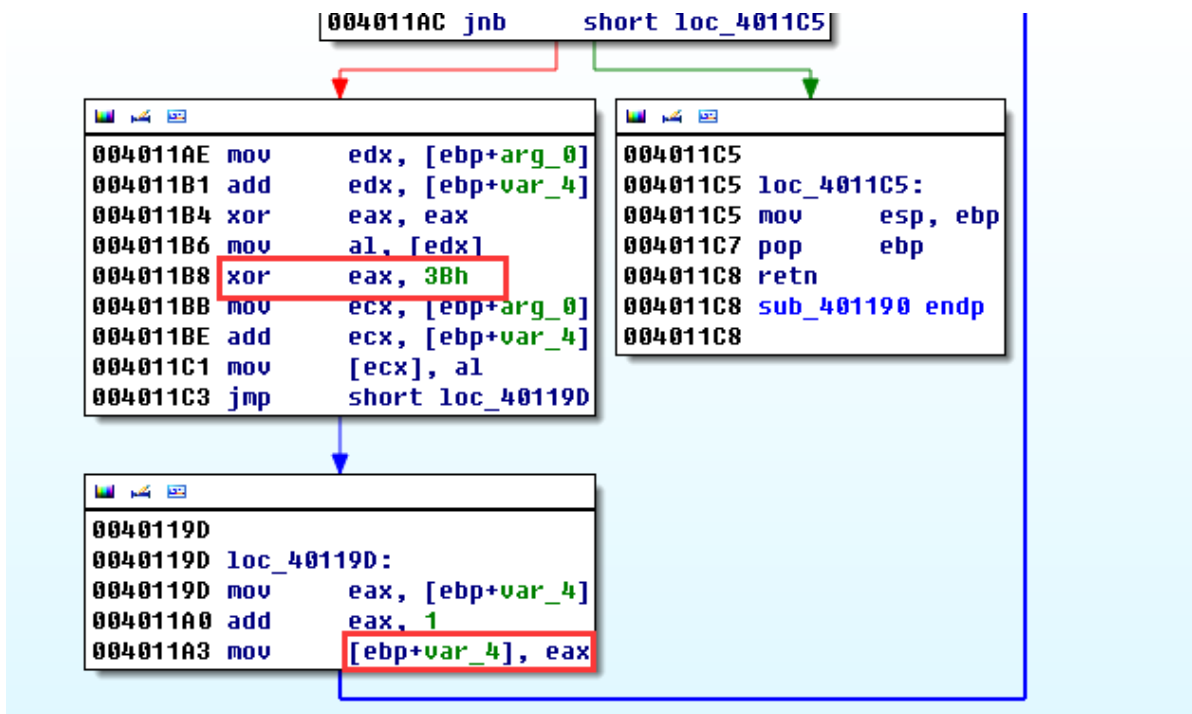
Address	Function	Instruction	
.text:00401007	sub_401000	xor	ecx, ecx
.text:0040101C	sub_401000	xor	edx, edx
.text:00401029	sub_401000	xor	ecx, ecx
.text:0040104E	sub_401000	xor	eax, eax
.text:0040105C	sub_401000	xor	edx, edx
.text:0040108D	sub_401000	xor	ecx, ecx
.text:004011B4	sub_401190	xor	eax, eax
.text:004011B8	sub_401190	xor	eax, 3Bh
.text:004011D6	sub_4011C9	xor	eax, eax
.text:004012A2	sub_4011C9	xor	al, al
.text:004012E6	sub_4011C9	xor	al, al
.text:004012FA	sub_4011C9	xor	al, al
.text:00401332	sub_401300	xor	eax, eax
.text:00401350	sub_401300	xor	eax, eax
.text:0040138E	sub_401300	xor	eax, eax
.text:00401463	_main	xor	eax, eax
.text:004021E5		xor	ecx, ecx
.text:00402202		xor	edx, edx
.text:00402BE2		xor	dh, [eax]
.text:00402BE6		xor	[eax], dh

可以发现非常多的地方都出现了xor，但是大部分都是自身进行异或，也就是清空，那么这些对于加密是没有任何作用的，所以这里忽略这些，之后我们发现还剩下3个地方需要注意

Address	Function	Instruction	
.text:00401007	sub_401000	xor	ecx, ecx
.text:0040101C	sub_401000	xor	edx, edx
.text:00401029	sub_401000	xor	ecx, ecx
.text:0040104E	sub_401000	xor	eax, eax
.text:0040105C	sub_401000	xor	edx, edx
.text:0040108D	sub_401000	xor	ecx, ecx
.text:004011B4	sub_401190	xor	eax, eax
.text:004011B8	sub_401190	xor	eax, 3Bh
.text:004011D6	sub_4011C9	xor	eax, eax
.text:004012A2	sub_4011C9	xor	al, al
.text:004012E6	sub_4011C9	xor	al, al
.text:004012FA	sub_4011C9	xor	al, al
.text:00401332	sub_401300	xor	eax, eax
.text:00401350	sub_401300	xor	eax, eax
.text:0040138E	sub_401300	xor	eax, eax
.text:00401463	_main	xor	eax, eax
.text:004021E5		xor	ecx, ecx
.text:00402202		xor	edx, edx
.text:00402BE2		xor	dh, [eax]
.text:00402BE6		xor	[eax], dh

发现在第一处他异或的是3Bh，猜测这里可能是使用了异或进行加密，可能密钥就是3Bh

双击点过去可以看见：

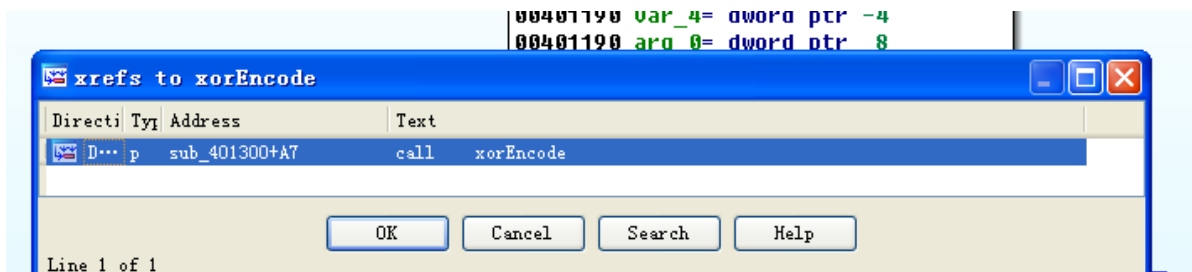


这里就是一个循环进行异或的操作，每次递增的是var_4中的内容（在这里就是长度），异或的内容是arg_0，那么这里很明显就是一个异或加密的操作了，密钥是3Bh，所以这个sub_401190函数就是进行异或加密的函数

问题3

恶意代码使用什么密钥加密，加密了什么内容？

根据刚刚的分析可以知道这个异或加密的密钥是3Bh



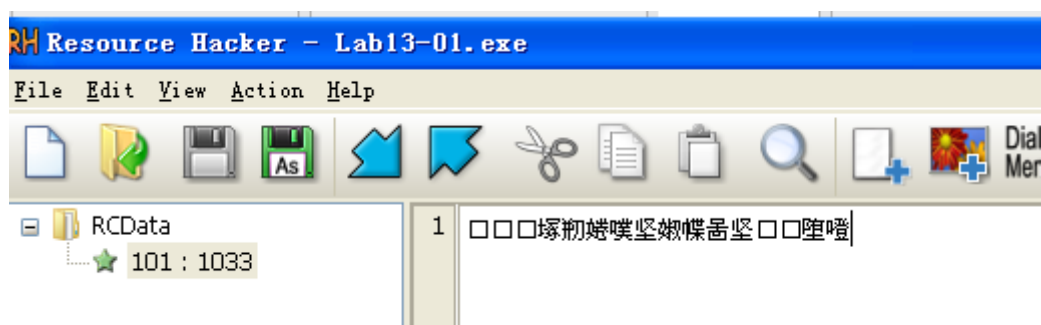
通过对刚刚这个函数的交叉引用可以发现只有一个地方调用了这个函数，那么很明显这里就是要加密的内容了，跳转到调用的位置

```

0040135B mov     edx, [ebp+hModule]
0040135E push    edx                ; hModule
0040135F call    ds:SizeofResource
00401365 mov     [ebp+dwBytes], eax
00401368 mov     eax, [ebp+dwBytes]
0040136B push    eax                ; dwBytes
0040136C push    40h                ; uFlags
0040136E call    ds:GlobalAlloc
00401374 mov     [ebp+var_4], eax
00401377 mov     ecx, [ebp+hResInfo]
0040137A push    ecx                ; hResInfo
0040137B mov     edx, [ebp+hModule]
0040137E push    edx                ; hModule
0040137F call    ds:LoadResource
00401385 mov     [ebp+hResData], eax
00401388 cmp     [ebp+hResData], 0
0040138C jnz     short loc_401392

```

可以看见在异或操作之前，这里释放了资源节的内容，那么很明显这个异或操作就是对资源节中的内容进行解密。



通过resource_hacker可以看见资源节里是一串乱码，更加印证了刚刚我们猜测他是对资源节中的内容进行解密了

00007050	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00007060	4C 4C 4C 15 4B 49 5A 58 4F 52 58 5A 57 56 5A 57	LL.KIZXORXZWVZW
00007070	4C 5A 49 5E 5A 55 5A 57 42 48 52 48 15 58 54 56	LZI^ZUZWBHRH.XTV
00007080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

使用winhex工具定位到刚刚的位置，可以看见刚刚的乱码，对其进行异或操作得到：

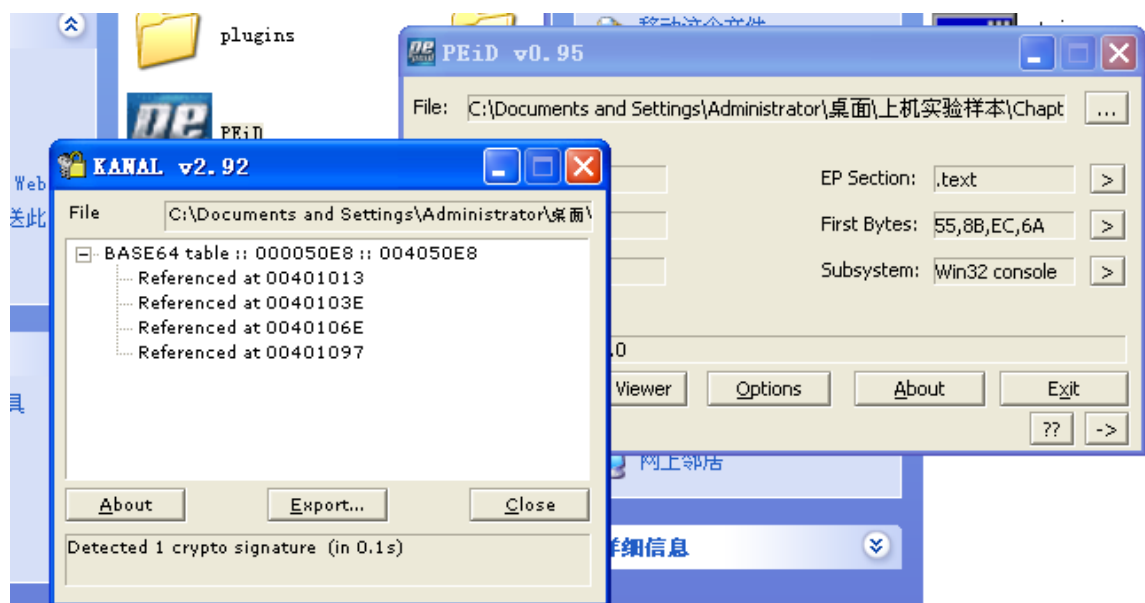
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
77 77 77 2E 70 72 61 63 74 69 63 61 6C 6D 61 6C	www.practicalmal
77 61 72 65 61 6E 61 6C 79 73 69 73 2E 63 6F 6D	wareanalysis.com
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

可以看见这里就是刚刚wireshark中抓到的那个url

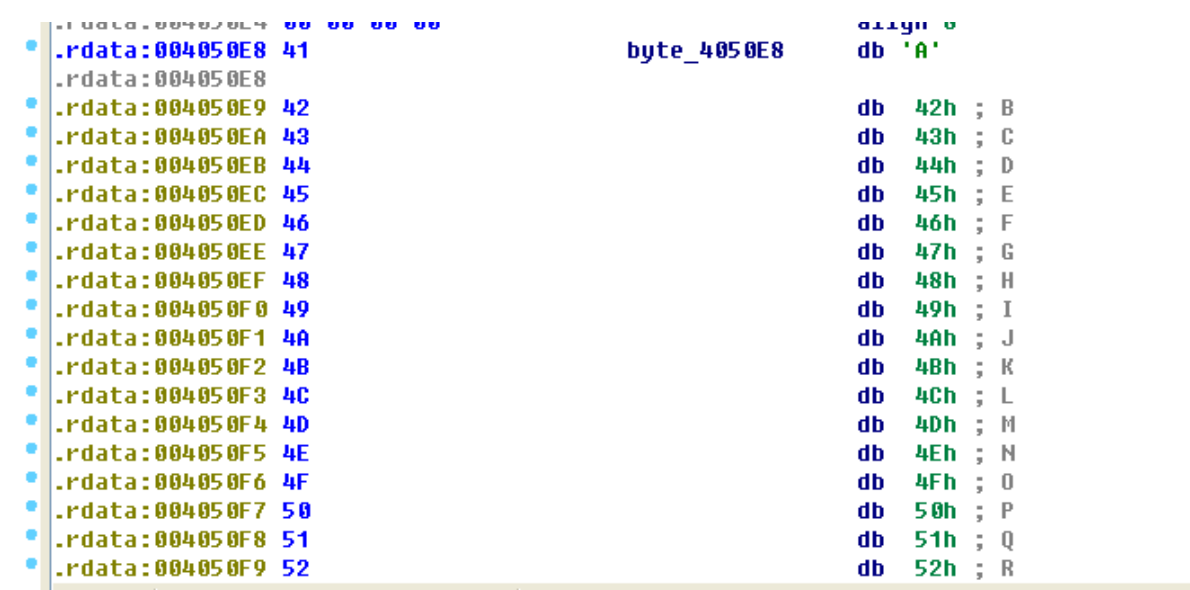
问题4

使用静态工具FindCrypt2、Krypto ANALyzer(KANAL)以及IDA插件识别一些其他类型的加密机制，你发现了什么？

使用PEiD中的插件KANAL可以看见



在如图中所示的几个位置还存在有Base64加密

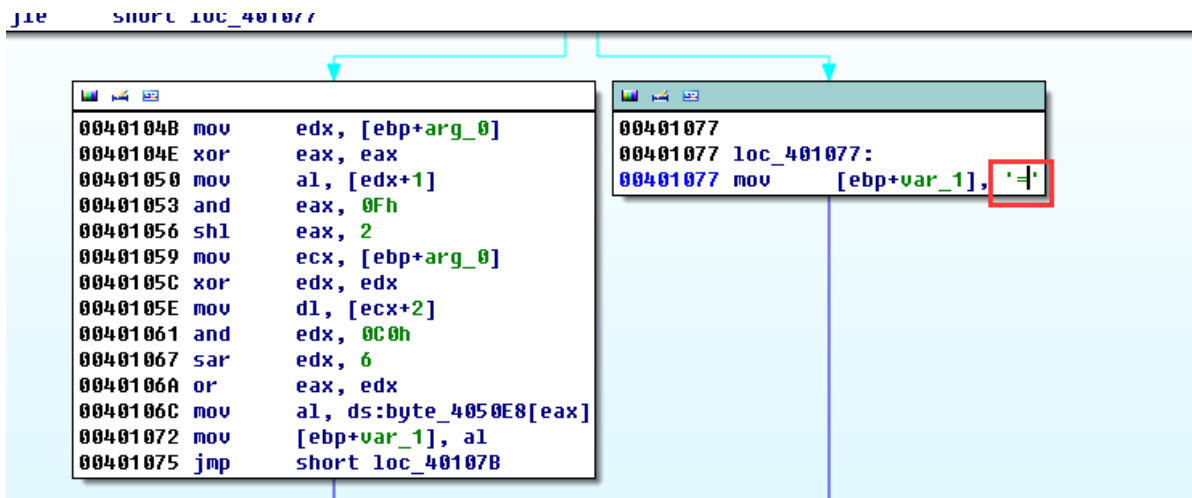


定位到位置上可以发现这里是一个标准的base64加密

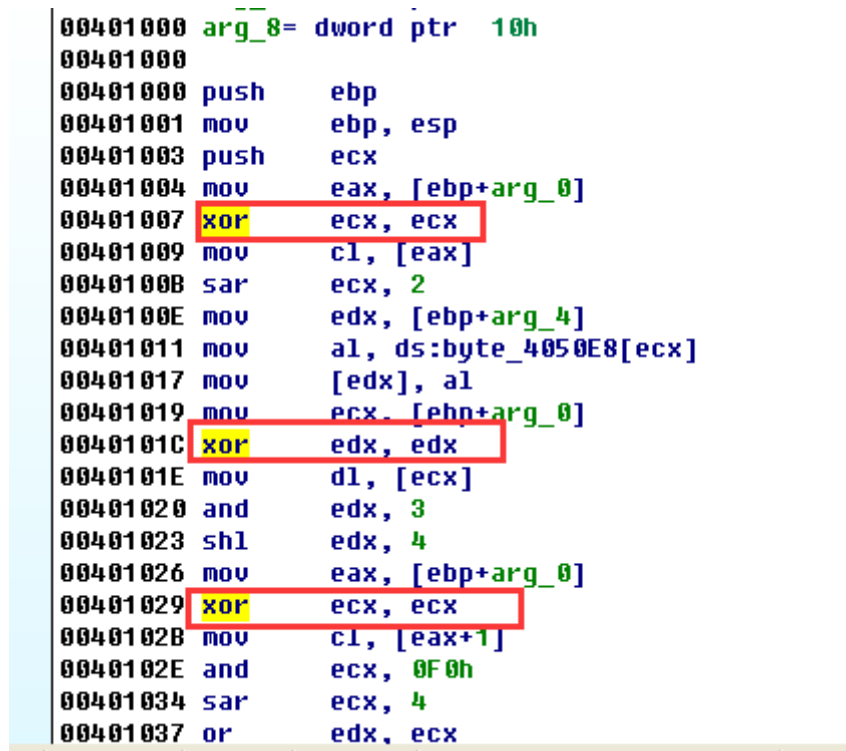
问题5

什么类型的加密被恶意代码用来发送部分网络流量？

通过刚刚我们分析的可以知道，url是使用的xor进行加密，还有一个就是GET请求我们没有找到，那么GET请求想来就是使用的这个base64进行加密



进入到其交叉引用的地方可以看见这里出现了=，也就是标准base64编码使用的填充符号

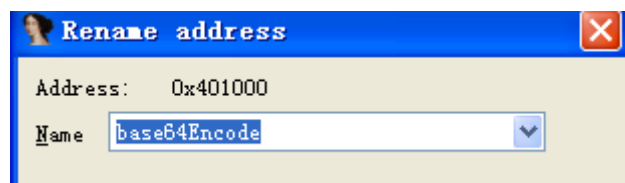


同时上面我们可以注意到这里一共进行了三次的清空寄存器和存放的操作，很明显这里就是对"GET"字样进行一个解密了

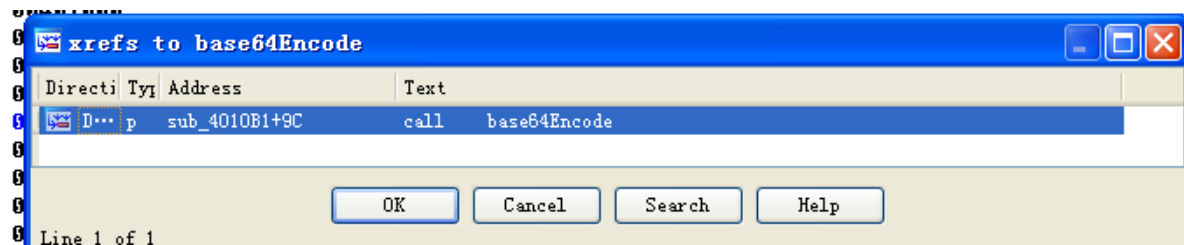
问题6

Base64编码函数在反汇编的何处？

通过交叉引用可以发现刚刚标准的base64编码在401000位置被引用



我们将其改名为base64Encode，再次通过交叉应用可以发现



是在4010B1+9C的位置被调用

问题7

恶意代码发送的Base64加密数据的最大长度是什么?加密了什么内容?

使用F5键进入到C语言代码的形式

```
signed int j; // [sp+8h] [bp-14h]@10
char v7[4]; // [sp+Ch] [bp-10h]@5
char v8[4]; // [sp+10h] [bp-Ch]@10
size_t v9; // [sp+14h] [bp-8h]@1
size_t v10; // [sp+18h] [bp-4h]@1

result = strlen(a1);
v9 = result;
v10 = 0;
v4 = 0;
while ( (signed int)v10 < (signed int)v9 )
{
    v3 = 0;
    for ( i = 0; i < 3; ++i )
    {
        v7[i] = a1[v10];
        result = v10;
        if ( (signed int)v10 >= (signed int)v9 )
        {
            result = i;
            v7[i] = 0;
        }
        else
        {
            v10++;
        }
    }
}
```

可以看见这个循环的判定条件是v10和v9的大小，并且v10的初始值是0，在之后的循环体内部v10会执行一个自增的操作，那么这个v10其实也就是相当于一个下角标的作用，来控制循环次数。并且v9字符串的初始值设置的是 `strlen(a1)`，也就是a1字符串的长度，那么也就是说这里循环就是遍历了整个字符串。

查看循环体内部


```

v3 = 0;
for ( i = 0; i < 3; ++i )
{
    v7[i] = a1[v10];
    result = v10;
    if ( (signed int)v10 >= (signed int)v9 )
    {
        result = i;
        v7[i] = 0;
    }
    else
    {
        ++v3;
        ++v10;
    }
}
if ( v3 )
{
    result = base64Encode(v7, v8, v3);
    for ( j = 0; j < 4; ++j )
    {
        result = j;
        *(_BYTE *)(v4++ + a2) = v8[j];
    }
}
return result;

```

可以看见循环体内部就是每次取出来三个字符，然后利用base64进行解密的操作

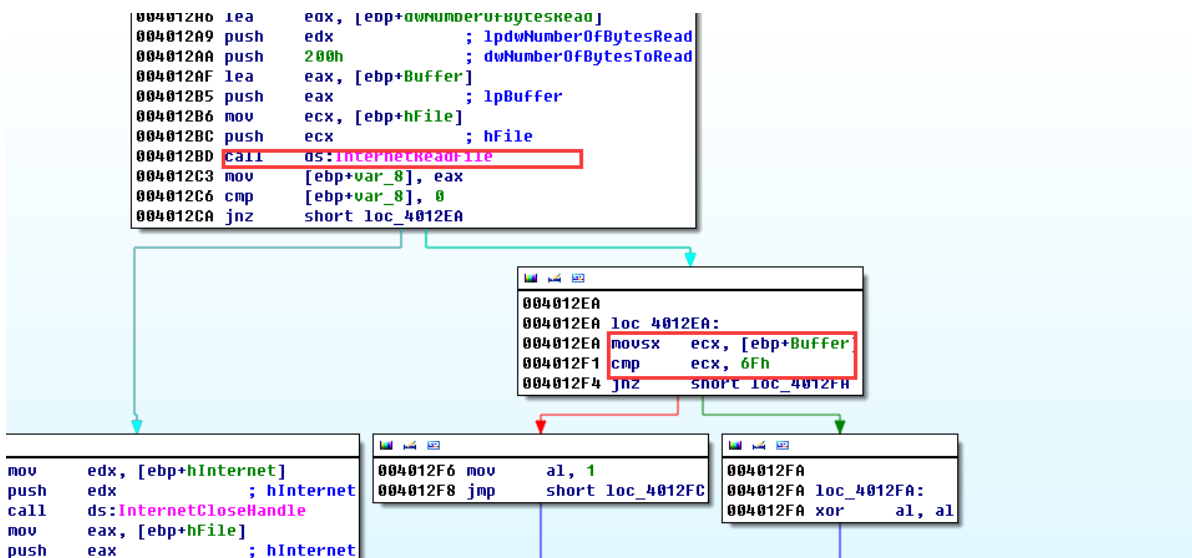
回到刚刚的函数体，查看一下交叉引用

```

004011E1 mov     [ebp+var_23], eax
004011E4 push    offset aMozilla4_0 ; "Mozilla/4.0"
004011E9 lea     ecx, [ebp+szAgent]
004011EC push    ecx                ; char *
004011ED call     _sprintf
004011F2 add     esp, 8
004011F5 push    100h                ; namelen
004011FA lea     edx, [ebp+name]
00401200 push    edx                ; name
00401201 call     gethostname
00401206 mov     [ebp+var_4], eax
00401209 push    0Ch                ; size_t
0040120B lea     eax, [ebp+name]
00401211 push    eax                ; char *
00401212 lea     ecx, [ebp+var_18]
00401215 push    ecx                ; char *
00401216 call     _strncpy
0040121B add     esp, 0Ch
0040121E mov     [ebp+var_C], 0
00401222 lea     edx, [ebp+var_30]
00401225 push    edx                ; int
00401226 lea     eax, [ebp+var_18]
00401229 push    eax                ; char *
0040122A call     callBase64
0040122F add     esp, 8

```

可以看见传递给这个函数的参数有两个，其中有一个来自上面的strncpy，这个函数的内容来自上面的gethostname的前12个字节。那么也就是说这里给到的是前12个字符。

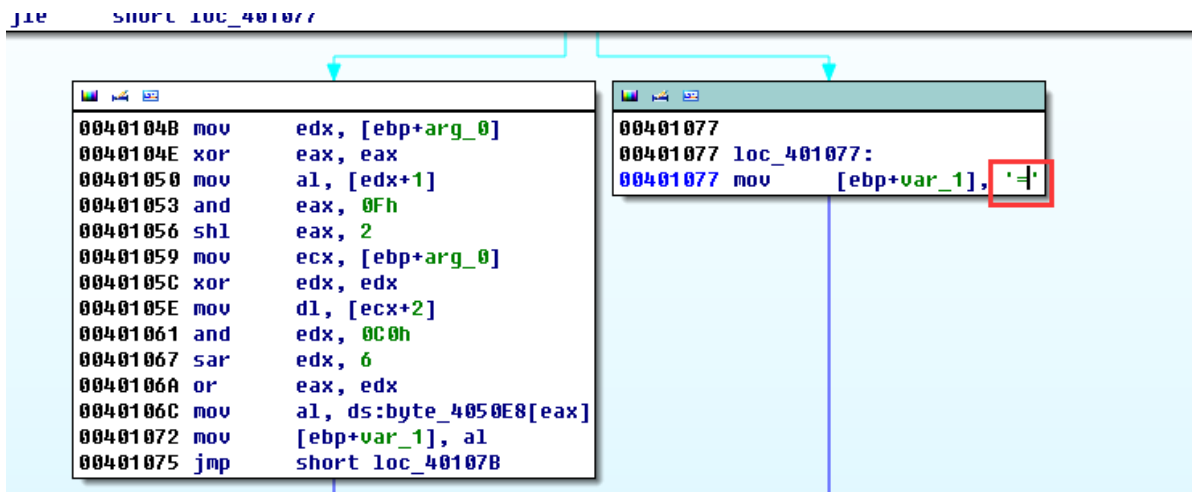


之后可以看见他获取了网络上的资源之后并对第一个字符进行比较，如果第一个字符是o则返回1，否则返回0

问题8

恶意代码中，你是否在Base64加密数据中看到了填充字符(=或者==)？

通过问题5中的分析我们可以看见



这里确实是使用了=进行填充，但是这里其实是当长度不是3的倍数或者是小于12个字符的时候才会填充，而这里因为之前取的就是12，所以虽然有填充在这里的，但是在本段恶意代码中不会发生。

问题9

这个恶意代码做了什么？

根据刚刚的分析我们可以知道，他会获得当前主机的名称并进行加密，然后向url发送一个GET请求并等待回复（判断回复的第一个字符是否为o），当接收到特定的回应之后才会退出。

lab 13-02

问题1

使用动态分析，确定恶意代码创建了什么？

使用procmon工具进行监控

Process Name	PID	Operation	Path	Result	Detail
Lab13-02.exe	4068	QueryNameIn...	C:\Documents and Settings\Administrator\桌面\上机实验样本\Chapter_13L\Lab13-02.exe	SUCCESS	Name: \Docume...
Lab13-02.exe	4068	QueryNameIn...	C:\Documents and Settings\Administrator\桌面\上机实验样本\Chapter_13L\Lab13-02.exe	SUCCESS	Name: \Docume...
Lab13-02.exe	4068	CreateFile	C:\WINDOWS\Prefetch\LAB13-02.EXE-00B1F5B.pf	NAME NOT FOUND	Desired Acces...
Lab13-02.exe	4068	CreateFile	C:\Documents and Settings\Administrator\桌面\上机实验样本\Chapter_13L	SUCCESS	Desired Acces...
Lab13-02.exe	4068	FileSystemC...	C:\Documents and Settings\Administrator\桌面\上机实验样本\Chapter_13L	SUCCESS	Control: FSCT...
Lab13-02.exe	4068	QueryOpen	C:\Documents and Settings\Administrator\桌面\上机实验样本\Chapter_13L\Lab13-02.exe.Local	NAME NOT FOUND	
Lab13-02.exe	4068	ReadFile	C:\Documents and Settings\Administrator\桌面\上机实验样本\Chapter_13L\Lab13-02.exe	SUCCESS	Offset: 24, 57...
Lab13-02.exe	4068	QueryOpen	C:\WINDOWS\system32\imm32.dll	SUCCESS	CreationTime:...
Lab13-02.exe	4068	CreateFile	C:\WINDOWS\system32\imm32.dll	SUCCESS	Desired Acces...
Lab13-02.exe	4068	CreateFile	C:\WINDOWS\system32\imm32.dll	SUCCESS	Desired Acces...
Lab13-02.exe	4068	QueryStanda...	C:\WINDOWS\system32\imm32.dll	SUCCESS	SyncType: Syn...
Lab13-02.exe	4068	QueryStanda...	C:\WINDOWS\system32\imm32.dll	SUCCESS	AllocationSiz...
Lab13-02.exe	4068	CloseFile	C:\WINDOWS\system32\imm32.dll	SUCCESS	SyncType: Syn...
Lab13-02.exe	4068	QueryOpen	C:\WINDOWS\system32\imm32.dll	SUCCESS	CreationTime:...
Lab13-02.exe	4068	CreateFile	C:\WINDOWS\system32\imm32.dll	SUCCESS	Desired Acces...
Lab13-02.exe	4068	CreateFile	C:\WINDOWS\system32\imm32.dll	SUCCESS	Desired Acces...
Lab13-02.exe	4068	QueryStanda...	C:\WINDOWS\system32\imm32.dll	SUCCESS	SyncType: Syn...
Lab13-02.exe	4068	QueryStanda...	C:\WINDOWS\system32\imm32.dll	SUCCESS	AllocationSiz...
Lab13-02.exe	4068	CreateFile	C:\WINDOWS\system32\imm32.dll	SUCCESS	SyncType: Syn...
Lab13-02.exe	4068	CloseFile	C:\WINDOWS\system32\imm32.dll	SUCCESS	
Lab13-02.exe	4068	QueryOpen	C:\WINDOWS\system32\imm32.dll	SUCCESS	CreationTime:...
Lab13-02.exe	4068	CreateFile	C:\WINDOWS\system32\imm32.dll	SUCCESS	Desired Acces...
Lab13-02.exe	4068	CreateFile	C:\WINDOWS\system32\imm32.dll	SUCCESS	Desired Acces...
Lab13-02.exe	4068	QueryOpen	C:\WINDOWS\system32\imm32.dll	SUCCESS	SyncType: Syn...
Lab13-02.exe	4068	QueryOpen	C:\WINDOWS\system32\imm32.dll	SUCCESS	CreationTime:...
Lab13-02.exe	4068	QueryOpen	C:\Documents and Settings\Administrator\桌面\上机实验样本\Chapter_13L\LPK.DLL	NAME NOT FOUND	CreationTime:...
Lab13-02.exe	4068	QueryOpen	C:\WINDOWS\system32\lpk.dll	SUCCESS	CreationTime:...
Lab13-02.exe	4068	CreateFile	C:\WINDOWS\system32\lpk.dll	SUCCESS	Desired Acces...
Lab13-02.exe	4068	CreateFile	C:\WINDOWS\system32\lpk.dll	SUCCESS	Desired Acces...
Lab13-02.exe	4068	CreateFile	C:\WINDOWS\system32\lpk.dll	SUCCESS	SyncType: Syn...
Lab13-02.exe	4068	CloseFile	C:\WINDOWS\system32\lpk.dll	SUCCESS	SyncType: Syn...
Lab13-02.exe	4068	QueryOpen	C:\Documents and Settings\Administrator\桌面\上机实验样本\Chapter_13L\USP10.dll	NAME NOT FOUND	CreationTime:...
Lab13-02.exe	4068	QueryOpen	C:\WINDOWS\system32\usp10.dll	SUCCESS	CreationTime:...
Lab13-02.exe	4068	CreateFile	C:\WINDOWS\system32\usp10.dll	SUCCESS	Desired Acces...
Lab13-02.exe	4068	CreateFile	C:\WINDOWS\system32\usp10.dll	SUCCESS	Desired Acces...
Lab13-02.exe	4068	CreateFile	C:\WINDOWS\system32\usp10.dll	SUCCESS	SyncType: Syn...
Lab13-02.exe	4068	CreateFile	C:\WINDOWS\system32\usp10.dll	SUCCESS	SyncType: Syn...
Lab13-02.exe	4068	CloseFile	C:\WINDOWS\system32\usp10.dll	SUCCESS	
Lab13-02.exe	4068	ReadFile	C:\Documents and Settings\Administrator\桌面\上机实验样本\Chapter_13L\Lab13-02.exe	SUCCESS	Offset: 4, 096...
Lab13-02.exe	4068	ReadFile	C:\Documents and Settings\Administrator\桌面\上机实验样本\Chapter_13L\Lab13-02.exe	SUCCESS	Offset: 28, 67...

可以看见在样本所在的目录创建了文件，并进行了写文件的操作

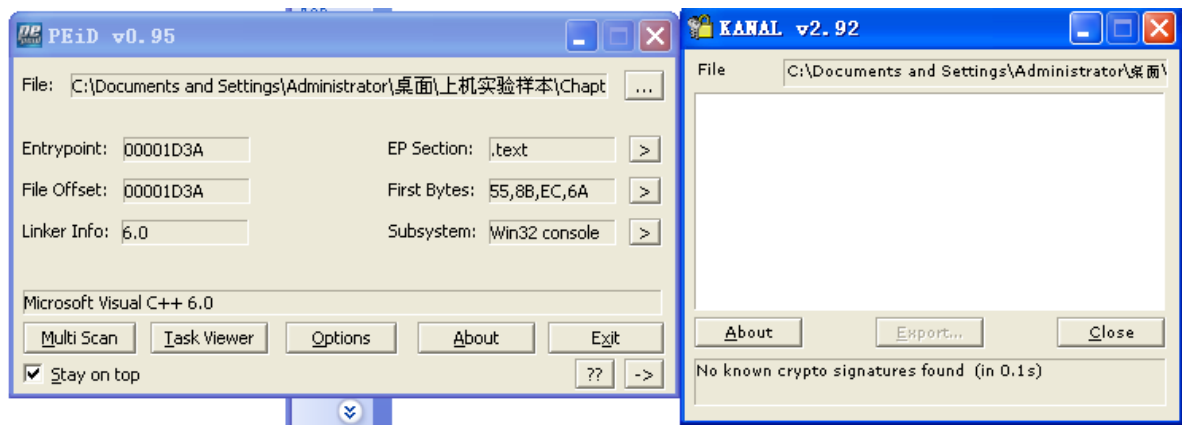


可以发现确实多出来了一些文件，可以发现这些文件的特点就是以temp开头，而且他们的大小都是6654KB

问题2

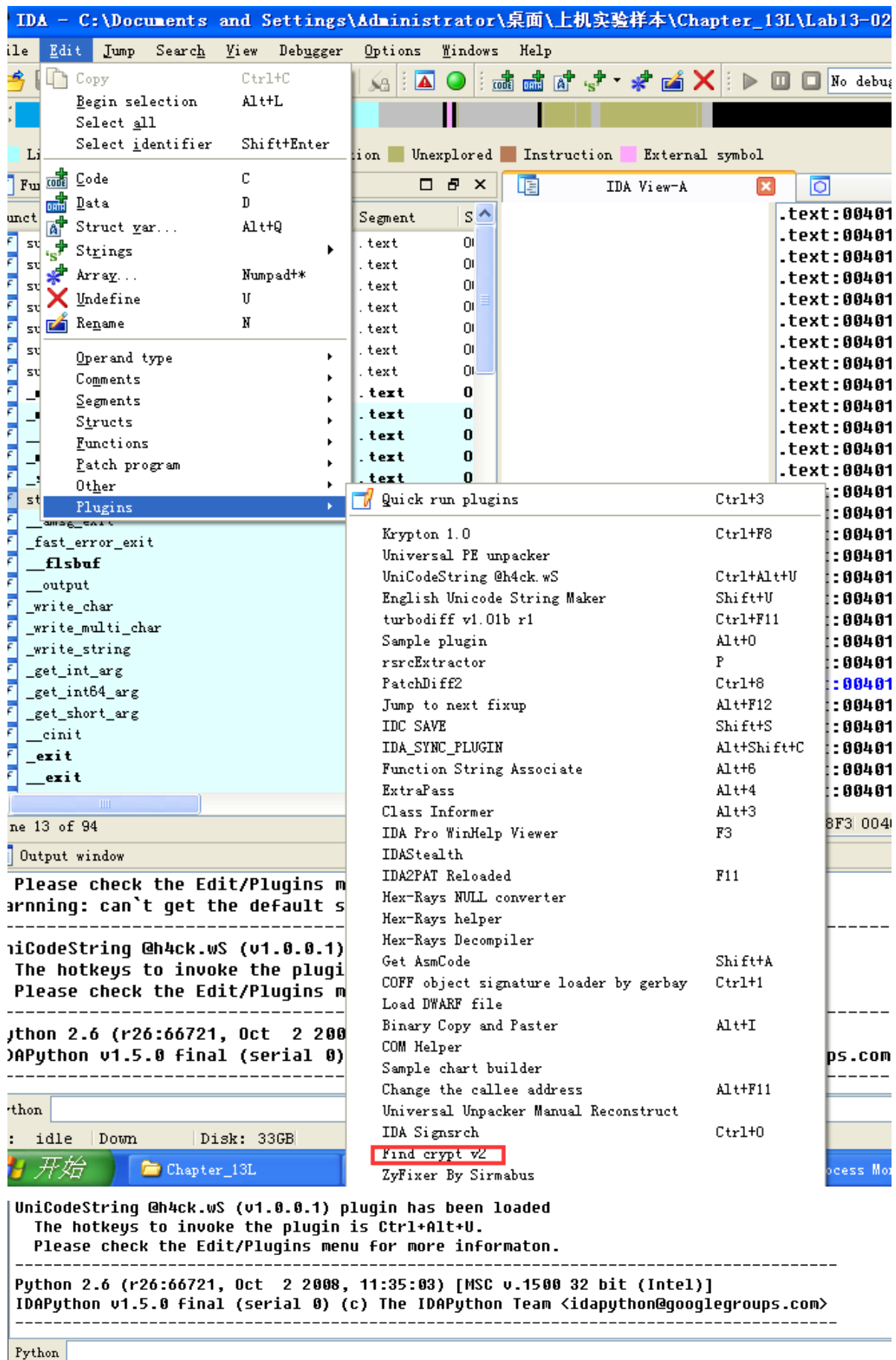
使用静态分析技术，例如xor指令的搜索、FindCrypt2、KANAL以及IDA熵插件，查找潜在的加密，你发现了什么？

首先使用KANAL查看



发现PEID中的插件并没有发现什么内容

之后使用IDA中的FindCrype插件查看



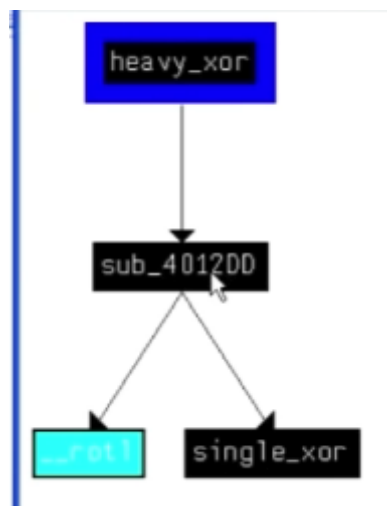
输出框依旧没有什么内容，也就是说这里没有找到东西

接下来搜索xor指令

Address	Function	Instruction	
.text:00401040	sub_401000	33 C0	xor eax, eax
.text:004012D6	sub_40128D	33 45 F0	xor eax, [ebp+var_10]
.text:0040171F		33 04 96	xor eax, [esi+edx*4]
.text:0040176F	sub_401739	33 11	xor edx, [ecx]
.text:0040177A	sub_401739	33 D1	xor edx, ecx
.text:00401785	sub_401739	33 D1	xor edx, ecx
.text:00401795	sub_401739	33 42 08	xor eax, [edx+8]
.text:004017A1	sub_401739	33 C2	xor eax, edx
.text:004017AC	sub_401739	33 C2	xor eax, edx
.text:004017BD	sub_401739	33 48 10	xor ecx, [eax+10h]
.text:004017C9	sub_401739	33 C8	xor ecx, eax
.text:004017D4	sub_401739	33 C8	xor ecx, eax
.text:004017E5	sub_401739	33 51 18	xor edx, [ecx+18h]
.text:004017F1	sub_401739	33 D1	xor edx, ecx
.text:004017FC	sub_401739	33 D1	xor edx, ecx
.text:0040191E	_main	33 C0	xor eax, eax
.text:0040311A		32 30	xor dh, [eax]
.text:0040311E		30 30	xor [eax], dh
.text:00403688		33 C9	xor ecx, ecx
.text:004036A5		33 D2	xor edx, edx

可以看见有很多条的xor指令，并且其中大部分来着sub_401739，还有一个条来自sub_40128D

通过查看交叉引用图可以发现



刚刚我们发现的使用多个xor指令的函数会调用一个4012DD函数，然后这个函数会调用40128D

问题3

基于问题1的解答，哪些导入函数将是寻找加密函数比较好的一个证据？

问题1中的特征我们发现是创建了很多具有一定特征的文件，而创建文件要使用的就是WriteFile函数，接下来我们针对这个函数的调用情况进行分析

xrefs to WriteFile			
Directi	Ty	Address	Text
Up	p	sub_401000+56	call ds:WriteFile
Up	p	__NMSG_WRITE+14A	call ds:WriteFile
Up	p	__write+D8	call ds:WriteFile
Up	p	__write+13D	call ds:WriteFile
Up	r	sub_401000+56	call ds:WriteFile
Up	r	__NMSG_WRITE+14A	call ds:WriteFile
Up	r	__write+D8	call ds:WriteFile
Up	r	__write+13D	call ds:WriteFile

Line 1 of 8

可以看见有两个地方被调用。那么由此也就可以认为WriteFile函数就是我们需要关注的函数

问题4

加密函数在反汇编的何处？

根据刚刚对WriteFile函数的分析我们可以知道主要是sub_401000函数调用了这个函数，查看一下这个函数

```
00401000 hObject= dword ptr -10h
00401000 NumberOfBytesWritten= dword ptr -0Ch
00401000 var_8= dword ptr -8
00401000 var_4= dword ptr -4
00401000 lpBuffer= dword ptr 8
00401000 nNumberOfBytesToWrite= dword ptr 0Ch
00401000 lpFileName= dword ptr 10h
00401000
```

可以看见这个函数的参数有如上几个，并且通过参数的名称就可以看出有一个参数是要写入的byte数量，一个是文件名，还有一个是缓冲区的指针。

找到调用这个函数的位置

```
00401888 call    ds:GetTickCount
0040188E mov     [ebp+var_4], eax
00401891 mov     ecx, [ebp+var_4]
00401894 push    ecx
00401895 push    offset aTemp08x ; "temp%08x"
0040189A lea     edx, [ebp+FileName]
004018A0 push    edx ; char *
004018A1 call    _sprintf
004018A6 add     esp, 0Ch
004018A9 lea     eax, [ebp+FileName]
004018AF push    eax ; lpFileName
004018B0 mov     ecx, [ebp+nNumberOfBytesToWrite]
004018B3 push    ecx ; nNumberOfBytesToWrite
004018B4 mov     edx, [ebp+lpBuffer]
004018B7 push    edx ; lpBuffer
004018B8 call    sub_401000
004018BD add     esp, 0Ch
004018C0 mov     eax, [ebp+lpBuffer]
```

看见出现了刚刚创建的文件的文件名的一部分，同时我们注意到，在上面有一个函数是GetTickCount，也就是获取系统启动的时间，那么猜测这里的文件名就是创建这样文件名的一个文件。

再往上可以发现：

```

00401851
00401851 push    ebp
00401852 mov     ebp, esp
00401854 sub     esp, 20Ch
0040185A mov     [ebp+lpBuffer], 0
00401861 mov     [ebp+nNumberOfBytesToWrite], 0
00401868 lea     eax, [ebp+nNumberOfBytesToWrite]
0040186B push    eax
0040186C lea     ecx, [ebp+lpBuffer]
0040186F push    ecx
00401870 call    sub_401070
00401875 add     esp, 8
00401878 mov     edx, [ebp+nNumberOfBytesToWrite]
0040187B push    edx
0040187C mov     eax, [ebp+lpBuffer]
0040187F push    eax
00401880 call    sub_40181F
00401885 add     esp, 8
00401888 call    ds:GetTickCount
0040188E mov     [ebp+var_4], eax

```

在上面先调用了另外两个函数，并且两个函数的参数都是指针和缓冲区的大小，那么根据逻辑顺序猜测第一个函数应该是获取文件，第二个函数是对文件进行加密或者解密的操作。

进入到第二个函数发现他就是刚刚调用xor函数的函数，并且密钥为10h

0040175E push edx	0040181B loc_40181B:
0040175F call sub_4012DD	0040181B mov esp, ebp
00401764 add esp, 4	0040181D pop ebp
00401767 mov eax, [ebp+arg_4]	0040181E retn
0040176A mov ecx, [ebp+arg_0]	0040181E sub_401739 endp
0040176D mov edx, [eax]	0040181E
0040176F xor edx, [ecx]	
00401771 mov eax, [ebp+arg_0]	
00401774 mov ecx, [eax+14h]	
00401777 shr ecx, 10h	
0040177A xor edx, ecx	
0040177C mov eax, [ebp+arg_0]	
0040177F mov ecx, [eax+0Ch]	
00401782 shl ecx, 10h	
00401785 xor edx, ecx	
00401787 mov eax, [ebp+arg_8]	
0040178A mov [eax], edx	
0040178C mov ecx, [ebp+arg_4]	
0040178F mov edx, [ebp+arg_0]	
00401792 mov eax, [ecx+4]	
00401795 xor eax, [edx+8]	
00401798 mov ecx, [ebp+arg_0]	
0040179B mov edx, [ecx+1Ch]	
0040179E shr edx, 10h	
004017A1 xor eax, edx	
004017A3 mov ecx, [ebp+arg_0]	
004017A6 mov edx, [ecx+14h]	

那么这个加密函数就在反汇编中的sub_401739

问题5

从加密函数追溯原始的加密内容，原始加密内容是什么？

根据刚刚的分析可以知道，两个函数中第二个函数是用来加密的，那么第一个函数就是获取要加密的文件的内容，对这个函数进行分析


```

0040107F call ds:GetSystemMetrics
00401085 mov [ebp+var_1C], eax
00401088 push 1 ; nIndex
0040108A call ds:GetSystemMetrics
00401090 mov [ebp+cy], eax
00401093 call ds:GetDesktopWindow
00401099 mov hWnd, eax
0040109E mov eax, hWnd
004010A3 push eax ; hWnd
004010A4 call ds:GetDC
004010AA mov hdc, eax
004010AF mov ecx, hdc
004010B5 push ecx ; hdc
004010B6 call ds:CreateCompatibleDC
004010BC mov [ebp+hdc], eax
004010BF mov edx, [ebp+cy]
004010C2 push edx ; cy
004010C3 mov eax, [ebp+var_1C]
004010C6 push eax ; cx
004010C7 mov ecx, hdc
004010CD push ecx ; hdc
004010CE call ds:CreateCompatibleBitmap
004010D4 mov [ebp+hbm], eax
004010D7 mov edx, [ebp+hbm]

```

发现这个函数调用了一串的系统函数，其中有一个GetDesktopWindow是用来获取桌面窗口的距离之后还有

```

push ecx ; hdc
call ds:BitBlt
lea edx, [ebp+pv]
push edx ; pv

004011B6 mov ecx, [ebp+hbm]
004011B9 push ecx ; hbm
004011BA mov edx, hdc
004011C0 push edx ; hdc
004011C1 call ds:GetDIBits
004011C7 mov eax, [ebp+dwBytes]
004011CA add eax, 36h
004011CD mov [ebp+var_74], eax

```

这两个函数用来获取位图的信息，并放到缓冲区。

经过网上资料的查询可以知道，这几个函数连用的作用就是获取用户桌面的信息，那么综合以上内容就知道其实这里获取的是用户桌面的内容，之后进行加密。

问题6

你是否能够找到加密算法?如果没有，你如何解密这些内容?

根据刚刚的分析，我们认为这个加密算法使用的就是一个简单的异或加密，对于异或加密操作来说，解密和加密是使用的同一套流程，所以解密的时候同样也是使用10h进行异或操作即可。

或者是在恶意代码获取了缓冲区的内容后，将缓冲区里的内容进行修改，改成以前得到的加密文件，然后让他再次执行xor的操作，就能够达到解密的效果

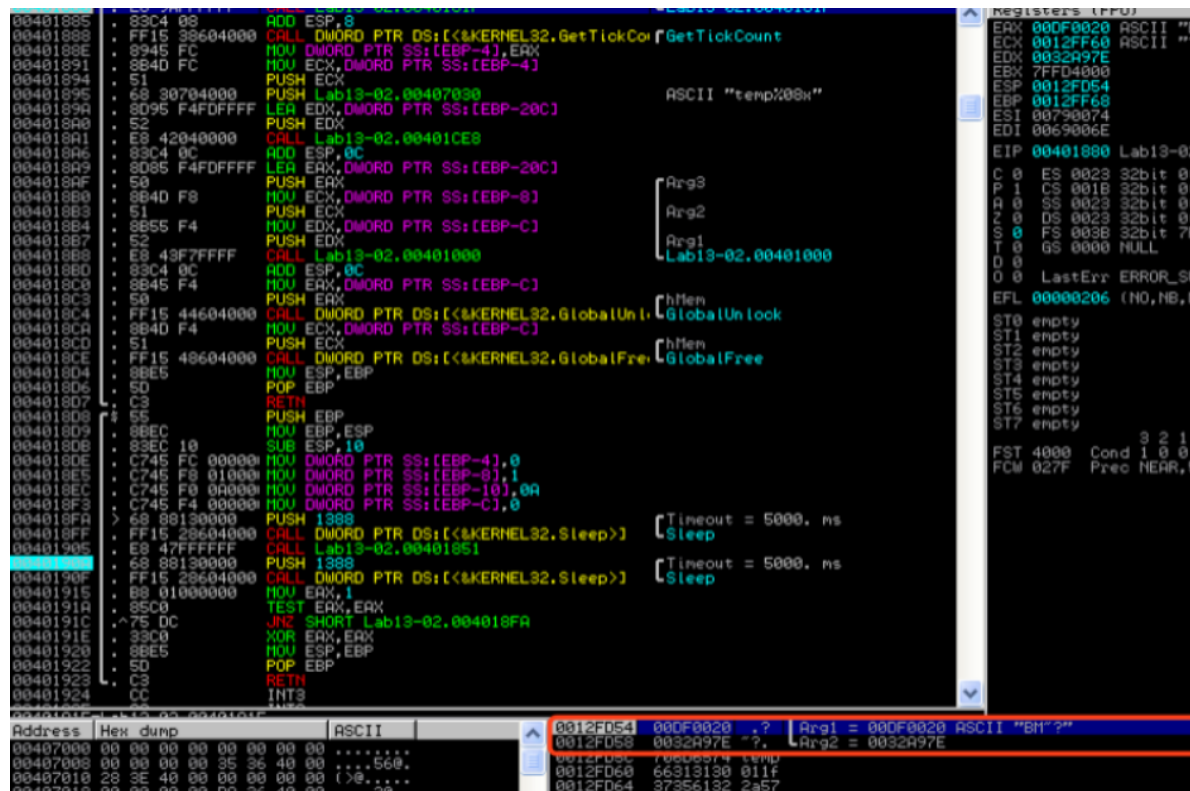
问题7

使用解密工具，你是否能够恢复加密文件中的一个文件到原始文件？

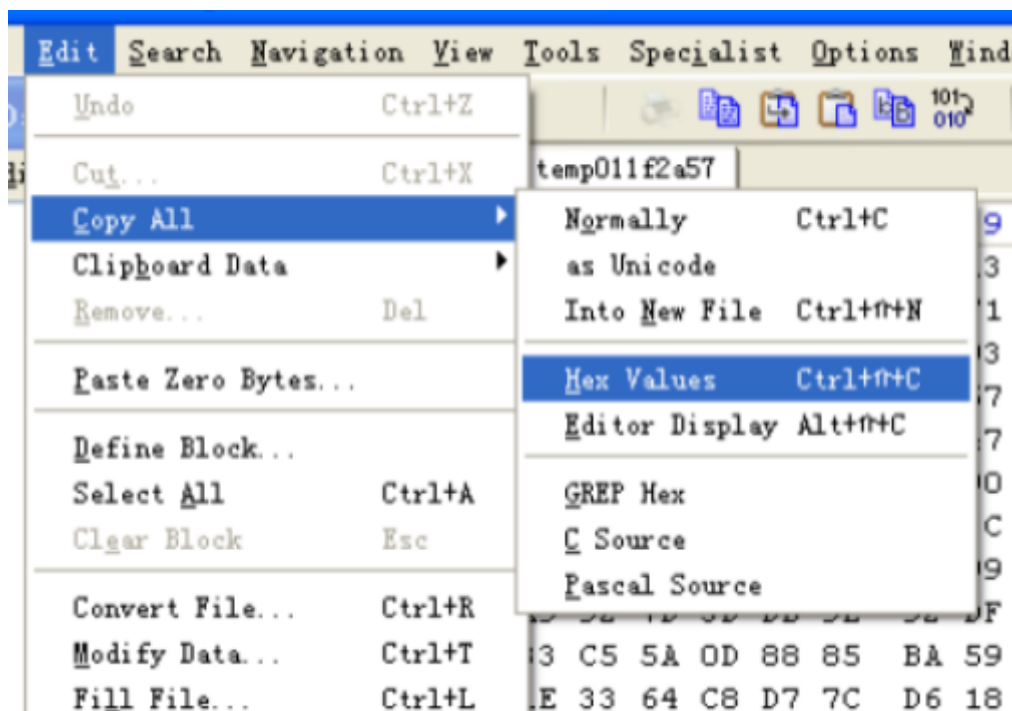
使用immunity工具进行中间内容的修改，首先载入程序，并在加密之前下一个断点（在ida中可以看见函数的位置为401880），然后在写入文件之后的位置再下一个断点（位置为：401905）。

想法是在到达他加密的位置时，将里面的内容替换为之前已经加密好的一个文件，然后执行加密操作（在这里就变成了解密），之后写入将解密的内容写入到新文件中，查看新文件即可。

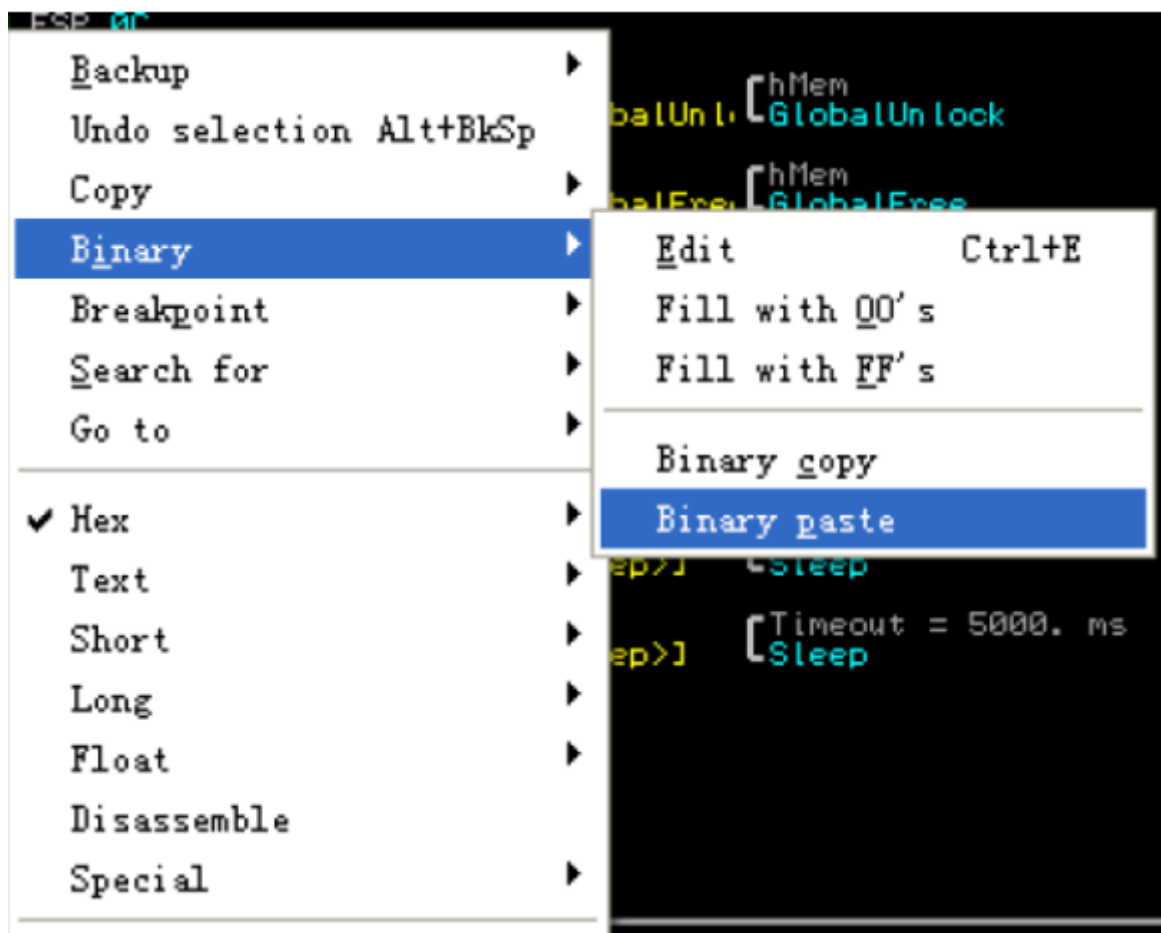
运行到第一个断点



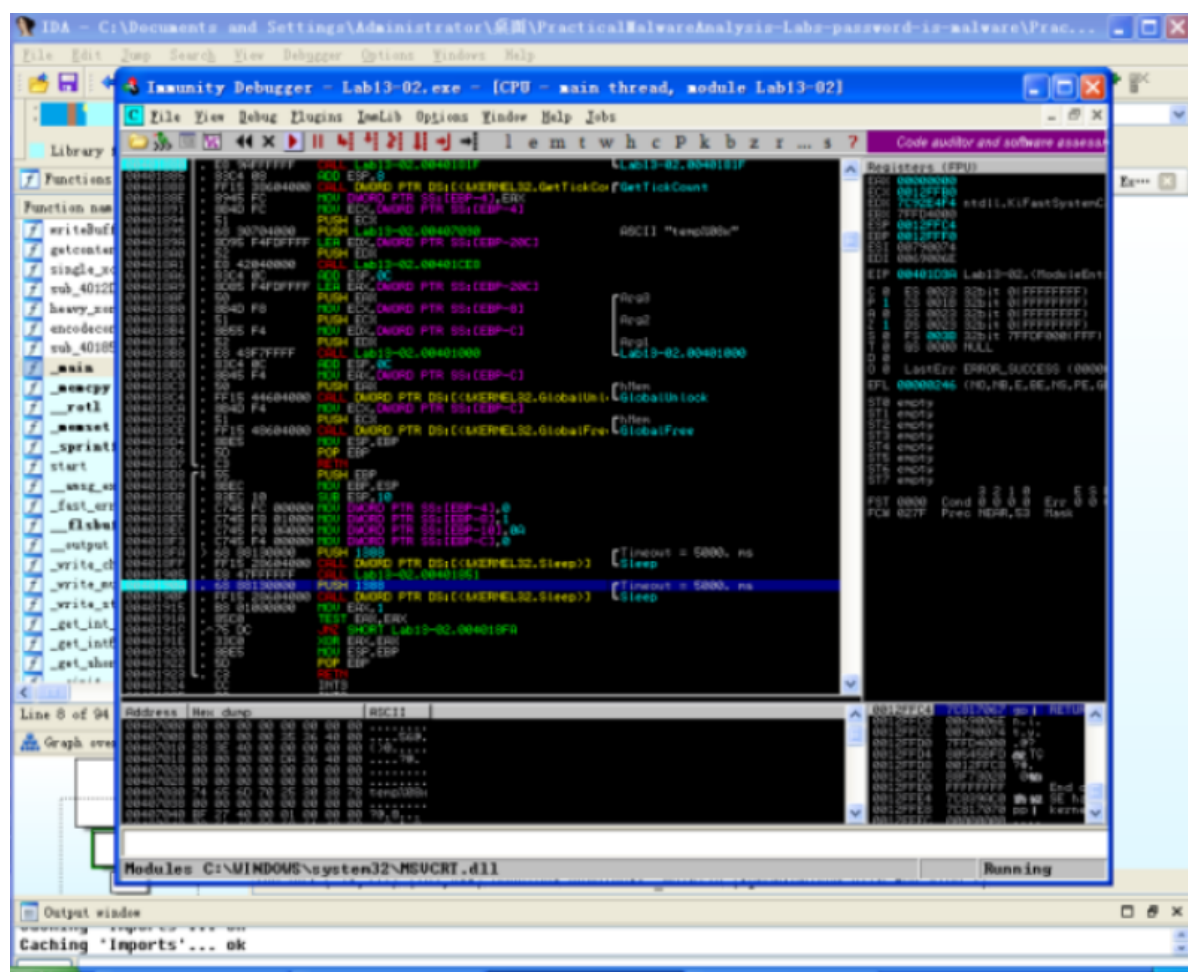
可以看见右下角这里就是即将加密的缓冲区和缓冲区的长度，在左边定位到缓冲区里的内容，然后使用winhex将我们要解密的文件以16进制的方式进行复制



再将缓冲区的内容全选以后进行替换



之后继续运行，程序运行结束之后得到一个新的文件。将这个文件的后缀改为.bmp之后可以双击打开



发现就是刚刚我们电脑状态的截图，解密成功

lab 13-03

问题1

比较恶意代码的输出字符串和动态分析提供的信息，通过这些比较，你发现哪些元素可能被加密？

首先使用strings工具简单查看一下有哪些字符串

```
cmd.exe
CloseHandle
CloseHandle
CloseHandle
CreateThread
CreateThread
ijklmnopqrstuvwxyz
www.practicalmalwareanalysis.com
L'H
d'A
Object not Initialized
Data not multiple of Block Size
Empty key
Incorrect key length
Incorrect block length
.?:AVexceptionee
.?:AVios_baseestdee
.?:AV?$basic_istream@DU?$char_traits@DEstddeestdee
.?:AV?$basic_ostream@DU?$char_traits@DEstddeestdee
.?:AV?$basic_streambuf@DU?$char_traits@DEstddeestdee
.?:AV?$basic_filebuf@DU?$char_traits@DEstddeestdee
.?:AV?$basic_istream@GU?$char_traits@GEstddeestdee
.?:AV?$basic_ostream@GU?$char_traits@GEstddeestdee
.?:AV?$basic_filebuf@GU?$char_traits@GEstddeestdee
.?:AVruntime_errorestdee
.?:AVfailure@ios_baseestdee
.?:AVfacet@localeestdee
.?:AV_Locimp@localeestdee
.?:AVlogic_errorestdee
.?:AVlength_errorestdee
.?:AVout_of_rangeestdee
.?:AVtype_infoee
```

发现在上面出现了一个url，在下面出现了有点像是乱码的字符串，但是好像又不是乱码

```
0K0
CDEFGHIJKLMNOPQRSTUVWXYZAbcdefghijklmnopqrstuvwxyzab0123456789+/-
ERROR: API = %s.
```

同时还可以注意到这里有一个类似于base64加密的字符串，发现下面的ERROR后面有一个=，还有格式化字符串，猜测会对这里进行一个加密

根据刚刚的分析我们可以认为这个程序有访问网络的行为，所以我们使用wireshark进行分析

4	2.74624200	192.168.159.131	192.168.159.2	DNS	92 Standard query 0x10b1	A www.practicalmalwareanalysis.com
5	3.00367500	192.168.159.1	192.168.159.255	UDP	305 Source port: 54915	Destination port: 54915
6	3.74692900	192.168.159.131	192.168.159.2	DNS	92 Standard query 0x10b1	A www.practicalmalwareanalysis.com
7	3.78898900	192.168.159.2	192.168.159.131	DNS	138 Standard query response 0x10b1	CNAME practicalmalwareanalysis.com A 192.0.78.25 A 192.0.78.24
8	3.78901400	192.168.159.2	192.168.159.131	DNS	138 Standard query response 0x10b1	CNAME practicalmalwareanalysis.com A 192.0.78.24 A 192.0.78.25
9	3.78985500	192.168.159.131	192.0.78.25	TCP	62 nlmreq > manyone-http [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1	
10	3.99630200	192.168.159.1	192.168.159.255	UDP	305 Source port: 54915	Destination port: 54915
11	4.99638500	192.168.159.1	192.168.159.255	UDP	305 Source port: 54915	Destination port: 54915
12	5.99434400	192.168.159.1	192.168.159.255	UDP	305 Source port: 54915	Destination port: 54915
13	6.99535200	192.168.159.1	192.168.159.255	UDP	305 Source port: 54915	Destination port: 54915
14	8.00155400	192.168.159.1	192.168.159.255	UDP	305 Source port: 54915	Destination port: 54915

Frame 4: 92 bytes on wire (736 bits), 92 bytes captured (736 bits) on interface 0
ethernet II, Src: vmware_22:21:3a (00:0c:29:22:21:3a), Dst: Vmware_fc:d2:b6 (00:50:56:fc:d2:b6)
Internet Protocol Version 4, Src: 192.168.159.131 (192.168.159.131), Dst: 192.168.159.2 (192.168.159.2)
User Datagram Protocol, Src Port: blackjack (1025), Dst Port: domain (53)
Domain Name System (Query)

```
00 50 56 fc d2 b6 00 0c 29 22 21 3a 08 00 45 00 .Pv.....}!...E.  
00 4e 05 00 00 80 11 75 aa c0 a8 9f 83 c0 a8 .N.....U.....  
9f 02 04 01 00 35 00 3a c3 50 10 b1 01 00 00 01 .....5.:.P.....  
00 00 00 00 00 00 03 77 77 77 18 70 72 61 63 74 .....www.pract  
69 63 61 6c 6d 61 6c 77 61 72 65 61 6e 61 6c 79 icalmalw areanaly  
73 69 73 03 63 6f 6d 00 01 00 01 0e 61 6c 79 sis.com. ....
```

可以看见出现了刚刚分析得到的url

问题2

使用静态分析搜索字符串xor来查找潜在的加密。通过这种方法，你发现什么类型的加密？

使用IDA查找xor指令以后可以发现

004023DB	sub_40223A	xor	edx, edx
004023F6	sub_40223A	xor	eax, eax
00402411	sub_40223A	xor	ecx, ecx
0040242C	sub_40223A	xor	edx, [ecx+0Ch]
00402484	sub_40223A	xor	edx, ds:Rijndael_Te1[ecx*4]
00402496	sub_40223A	xor	edx, ds:Rijndael_Te2[ecx*4]
004024A6	sub_40223A	xor	edx, ds:Rijndael_Te3[ecx*4]
004024B0	sub_40223A	xor	edx, [eax]
004024D4	sub_40223A	xor	eax, ds:Rijndael_Te1[edx*4]
004024E7	sub_40223A	xor	eax, ds:Rijndael_Te2[ecx*4]
004024F7	sub_40223A	xor	eax, ds:Rijndael_Te3[edx*4]
00402501	sub_40223A	xor	eax, [ecx+4]
00402525	sub_40223A	xor	ecx, ds:Rijndael_Te1[ecx*4]
00402538	sub_40223A	xor	ecx, ds:Rijndael_Te2[edx*4]
00402547	sub_40223A	xor	ecx, ds:Rijndael_Te3[ecx*4]
00402551	sub_40223A	xor	ecx, [edx+8]
00402575	sub_40223A	xor	edx, ds:Rijndael_Te1[ecx*4]
00402587	sub_40223A	xor	edx, ds:Rijndael_Te2[ecx*4]
00402597	sub_40223A	xor	edx, ds:Rijndael_Te3[ecx*4]
004025A1	sub_40223A	xor	edx, [eax+0Ch]
004025FB	sub_40223A	xor	eax, ecx
0040261A	sub_40223A	xor	ecx, edx
0040263B	sub_40223A	xor	edx, eax

有非常多的地方都使用了xor，一共有6个函数使用了xor，那么这些函数可能存在有加密的行为，对这6个函数都进行重命名，分别为x1到x6。那么根据刚刚的分析，猜测有6处都有加密的可能，并且使用的加密方式可能是异或

问题3

使用静态工具，如FindCrypt2、KANAL以及IDA插件识别一些其他类型的加密机制。发现的结果与搜索字符XOR结果比较如何？

使用IDA的FindCrypt2插件进行查找

```

The initial autoanalysis has been finished.
40CB08: found const array Rijndael_Te0 (used in Rijndael)
40CF08: found const array Rijndael_Te1 (used in Rijndael)
40D308: found const array Rijndael_Te2 (used in Rijndael)
40D708: found const array Rijndael_Te3 (used in Rijndael)
40DB08: found const array Rijndael_Td0 (used in Rijndael)
40DF08: found const array Rijndael_Td1 (used in Rijndael)
40E308: found const array Rijndael_Td2 (used in Rijndael)
40E708: found const array Rijndael_Td3 (used in Rijndael)
Found 8 known constant arrays in total.

```

可以发现找到了8处位置使用了加密算法，并且这个标注的Rijndael就是指的AES中的算法

分别查看一下这8个位置的交叉引用

1.

xrefs to Rijndael_Te0

Directi	Typ	Address	Text
Up	r	x2+243	mov edx, ds:Rijndael_Te0[ecx*4]
Up	r	x2+293	mov eax, ds:Rijndael_Te0[ecx*4]
Up	r	x2+2E4	mov ecx, ds:Rijndael_Te0[edx*4]
Up	r	x2+334	mov edx, ds:Rijndael_Te0[ecx*4]
Up	r	x4+1ED	mov ecx, ds:Rijndael_Te0[ecx*4]

2.

xrefs to Rijndael_Te1

Directi	Typ	Address	Text
Up	r	x2+24A	xor edx, ds:Rijndael_Te1[ecx*4]
Up	r	x2+29A	xor eax, ds:Rijndael_Te1[ecx*4]
Up	r	x2+2EB	xor ecx, ds:Rijndael_Te1[ecx*4]
Up	r	x2+33B	xor edx, ds:Rijndael_Te1[ecx*4]
Up	r	x4+1F4	xor ecx, ds:Rijndael_Te1[edx*4]

OK Cancel Search Help

3.

xrefs to Rijndael_Te2

Directi	Typ	Address	Text
Up	r	x2+25C	xor edx, ds:Rijndael_Te2[ecx*4]
Up	r	x2+2AD	xor eax, ds:Rijndael_Te2[ecx*4]
Up	r	x2+2FE	xor ecx, ds:Rijndael_Te2[edx*4]
Up	r	x2+34D	xor edx, ds:Rijndael_Te2[ecx*4]
Up	r	x4+218	xor ecx, ds:Rijndael_Te2[edx*4]

OK Cancel Search Help

4.

xrefs to Rijndael_Te3

Directi	Typ	Address	Text
Up	r	x2+26C	xor edx, ds:Rijndael_Te3
Up	r	x2+2BD	xor eax, ds:Rijndael_Te3
Up	r	x2+30D	xor ecx, ds:Rijndael_Te3
Up	r	x2+35D	xor edx, ds:Rijndael_Te3
Up	r	x4+239	xor ecx, ds:Rijndael_Te3

OK Cancel Search

5. xrefs to Rijndael_Id0

Directi	Type	Address	Text
Up	r	x3+248	mov edx, ds:Rijndael_Id0[eax]
Up	r	x3+298	mov eax, ds:Rijndael_Id0[ecx]
Up	r	x3+2E9	mov ecx, ds:Rijndael_Id0[edx]
Up	r	x3+339	mov edx, ds:Rijndael_Id0[eax]
Up	r	x5+1F0	mov ecx, ds:Rijndael_Id0[ecx]

OK Cancel Search

6. xrefs to Rijndael_Id1

Directi	Type	Address	Text
Up	r	x3+24F	xor edx, ds:Rijndael_Id1[ecx*4]
Up	r	x3+29F	xor eax, ds:Rijndael_Id1[edx*4]
Up	r	x3+2F0	xor ecx, ds:Rijndael_Id1[eax*4]
Up	r	x3+340	xor edx, ds:Rijndael_Id1[ecx*4]
Up	r	x5+1F7	xor ecx, ds:Rijndael_Id1[edx*4]

OK Cancel Search He

7. xrefs to Rijndael_Id2

Directi	Type	Address	Text
Up	r	x3+261	xor edx, ds:Rijn
Up	r	x3+2B2	xor eax, ds:Rijn
Up	r	x3+303	xor ecx, ds:Rijn
Up	r	x3+352	xor edx, ds:Rijn
Up	r	x5+21B	xor ecx, ds:Rijn

OK Cancel

Line 1 of 5

8. xrefs to Rijndael_Id3

Directi	Type	Address	Text
Up	r	x3+271	xor edx, ds:Rijndael_Id3[ecx*4]
Up	r	x3+2C2	xor eax, ds:Rijndael_Id3[edx*4]
Up	r	x3+312	xor ecx, ds:Rijndael_Id3[eax*4]
Up	r	x3+362	xor edx, ds:Rijndael_Id3[ecx*4]
Up	r	x5+23C	xor ecx, ds:Rijndael_Id3[edx*4]

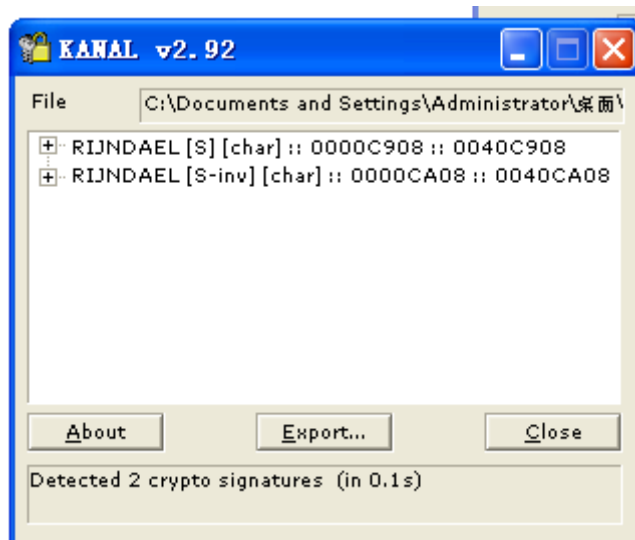
OK Cancel Search

Line 1 of 5

db 41h ; A

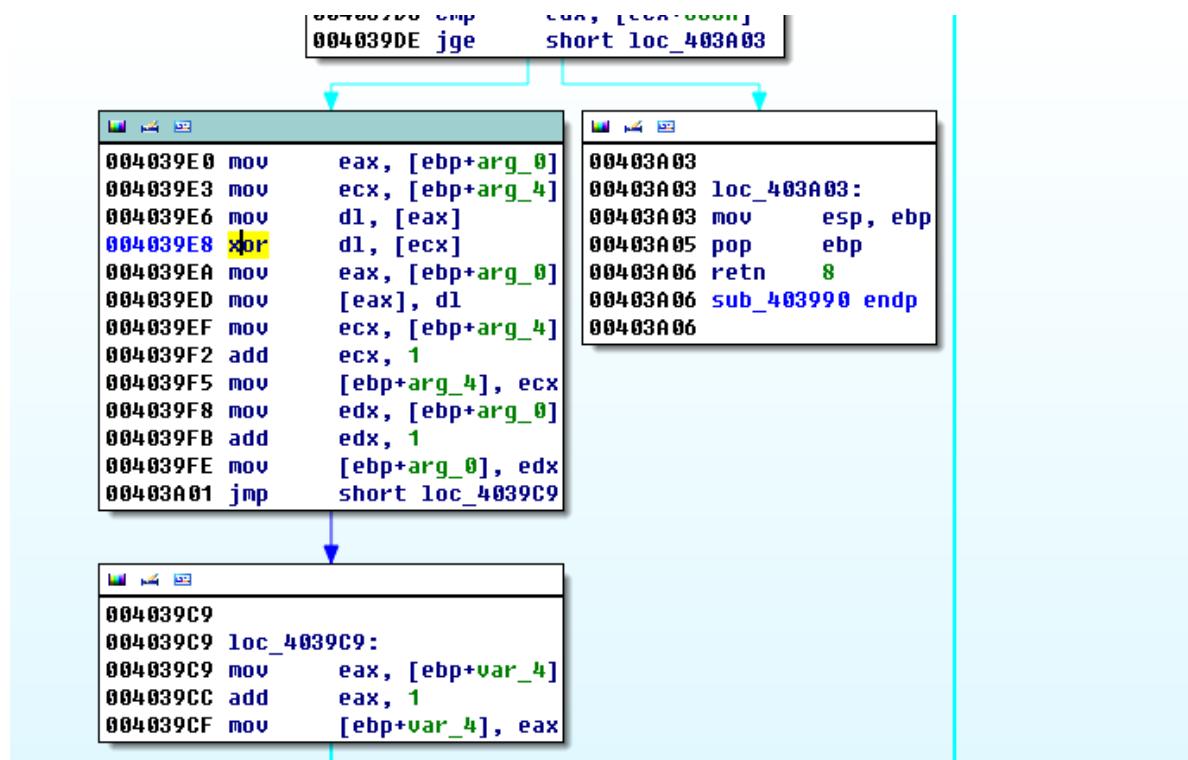
发现这8处一共出现了两种组合：3和5以及2和4，前4个地方使用2和4进行加密；后4个地方使用3和5进行解密

使用PEiD工具的插件可以发现



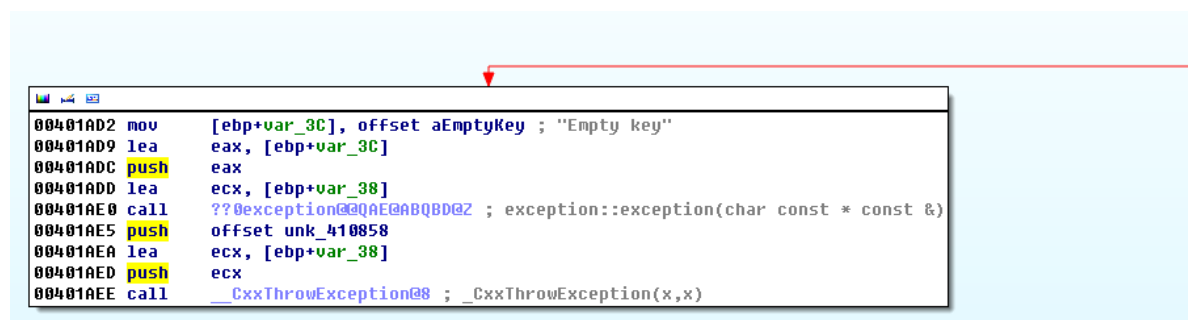
也是发现了两处位置进行了加解密的操作。那么根据刚刚的内容我们可以知道，3和5是为AES的解密；2和4是AES的加密

我们注意到x6和x1还没有被引用过，所以接下来先分析x6的作用是什么

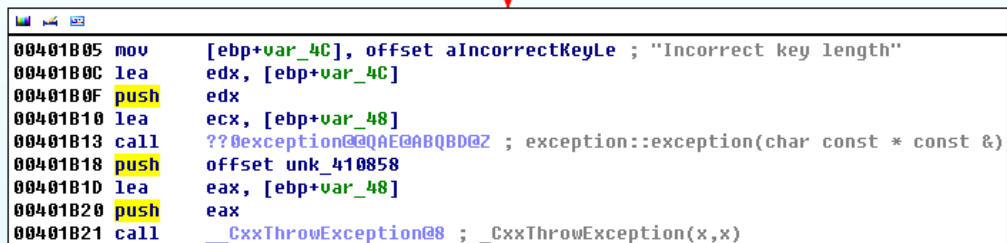


可以发现这里就是一个异或的加密操作，其中在异或的时候有两个参数，一个是arg0，也就是加密前的内容，还有一个是arg4，也就是加密后的内容。

再来看一下x1，



发现x1是首先判断一下密钥是否为空，如果是空的就返回一个提示信息



A screenshot of a debugger window showing assembly code. A red arrow points from the text above to the first instruction. The code is as follows:

```
00401B05 mov     [ebp+var_4C], offset aIncorrectKeyLe ; "Incorrect key length"
00401B0C lea     edx, [ebp+var_4C]
00401B0F push    edx
00401B10 lea     ecx, [ebp+var_48]
00401B13 call    ??0exception@@QAE@ABQBD@Z ; exception::exception(char const * const &)
00401B18 push    offset unk_410858
00401B1D lea     eax, [ebp+var_48]
00401B20 push    eax
00401B21 call    __CxxThrowException@8 ; _CxxThrowException(x,x)
```

然后检查密钥的长度是否符合要求，不符合的话同样返回一个提示信息

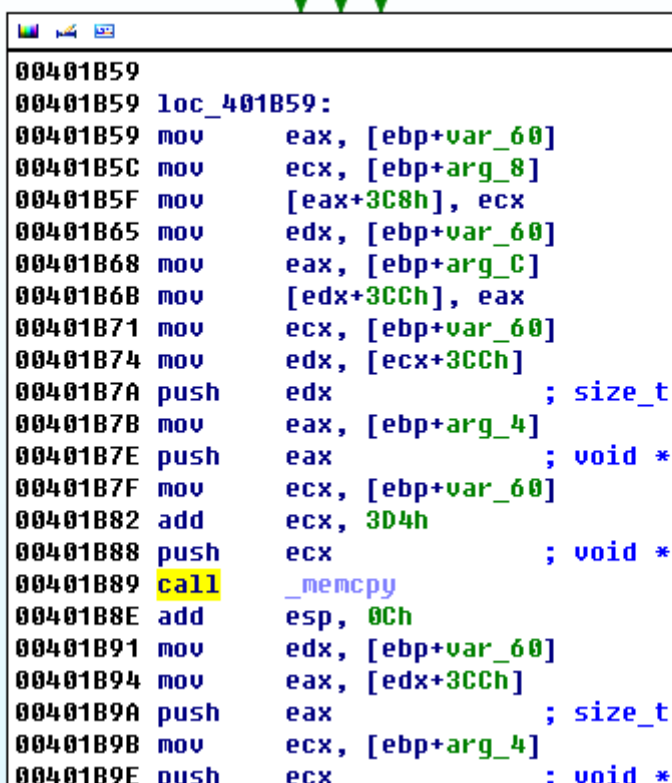


A screenshot of a debugger window showing assembly code. A red arrow points from the text above to the first instruction. The code is as follows:

```
00401B38 mov     [ebp+var_5C], offset aIncorrectBlock ; "Incorrect block length"
00401B3F lea     ecx, [ebp+var_5C]
00401B42 push    ecx
00401B43 lea     ecx, [ebp+var_58]
00401B46 call    ??0exception@@QAE@ABQBD@Z ; exception::exception(char const * const &)
00401B4B push    offset unk_410858
00401B50 lea     edx, [ebp+var_58]
00401B53 push    edx
00401B54 call    __CxxThrowException@8 ; _CxxThrowException(x,x)
```

最后检查的是块的长度，不符合的话返回一个提示信息

当这些检查都通过以后会执行后面的操作



A screenshot of a debugger window showing assembly code. Three green arrows point from the text above to the first three instructions. The code is as follows:

```
00401B59
00401B59 loc_401B59:
00401B59 mov     eax, [ebp+var_60]
00401B5C mov     ecx, [ebp+arg_8]
00401B5F mov     [eax+3C8h], ecx
00401B65 mov     edx, [ebp+var_60]
00401B68 mov     eax, [ebp+arg_C]
00401B6B mov     [edx+3CCh], eax
00401B71 mov     ecx, [ebp+var_60]
00401B74 mov     edx, [ecx+3CCh]
00401B7A push    edx ; size_t
00401B7B mov     eax, [ebp+arg_4]
00401B7E push    eax ; void *
00401B7F mov     ecx, [ebp+var_60]
00401B82 add     ecx, 3D4h
00401B88 push    ecx ; void *
00401B89 call    _memcpy
00401B8E add     esp, 0Ch
00401B91 mov     edx, [ebp+var_60]
00401B94 mov     eax, [edx+3CCh]
00401B9A push    eax ; size_t
00401B9B mov     ecx, [ebp+arg_4]
00401B9E push    ecx ; void *
```

查看x1的交叉应用可以发现

```

0040187H mov     esp, esp
0040187C sub     esp, 1ACh
00401882 push    10h          ; int
00401884 push    10h          ; int
00401886 push    offset unk_413374 ; void *
0040188B push    offset aijklmnopqrstuv ; "ijklmnopqrstuvwx"
00401890 mov     ecx, offset unk_412EF8
00401895 call    x1
0040189A lea     eax, [ebp+WSAData]
004018A0 push    eax          ; lpWSAData
004018A1 push    202h         ; wVersionRequested
004018A6 call    ds:WSAStartup
004018AC mov     [ebp+var_194], eax
004018B2 cmp     [ebp+var_194], 0
004018B9 jz      short loc_4018C5

```

这个函数是由main函数直接进行调用的，而且在这个函数被调用之前有一个unk_412ef8的一个引用；通过交叉引用可以发现这个还被其他位置进行了调用

```

00401429 mov     ecx, offset unk_412EF8
0040142E call    x6
00401433 push    0          ; lpOverlapped

```

可以看见这个偏移也被引入到了x6中，经过分析我们可以得知这个x6是AES的一个启动函数，所以这个其实就是AES要加密的一个对象。而刚刚那个x1对这个对象进行了一系列的检查，那么其实这个x1就是加密器的初始化函数。

问题4

恶意代码使用哪两种加密技术？

根据问题3中的分析可以知道为AES的Rijndael 算法和自定义的一个Base64加密技术

问题5

对于每一种加密技术，它们的密钥是什么？

根据之前的分析我们知道了x1就是AES的初始化函数，那么密钥就是在x1中进行了检查

```

00401AD2 mov     [ebp+var_3C], offset aEmptyKey ; "Empty key"
00401AD9 lea     eax, [ebp+var_3C]
00401ADC push    eax
00401ADD lea     ecx, [ebp+var_38]
00401AE0 call    ??0exception@@@AEB@ABQBD@Z ; exception::exception(char const * const &)
00401AE5 push    offset unk_410858
00401AEA lea     ecx, [ebp+var_38]
00401AED push    ecx
00401AEE call    _CxxThrowException@8 ; _CxxThrowException(x,x)

```

在x1中有一个地方会提示空密钥，而这个提示是基于arg0的

```

00401AC5 sub     esp, 00h
00401AC8 push    esi
00401AC9 mov     [ebp+var_60], ecx
00401ACB cmp     [ebp+arg_0], 0
00401AD0 jnz     short loc_401AF3

```

回到调用他的地方查看参数

```

00401882 push    10h                ; int
00401884 push    10h                ; int
00401886 push    offset unk_413374 ; void *
0040188B push    offset aijklmnopqrstuv ; "ijklmnopqrstuvwx"
00401890 mov     ecx, offset unk_412EF8
00401895 call    x1
0040189A lea     eax, [ebp+WSAData]
004018A0 push    eax                ; lpWSAData
004018A1 push    202h                : wVersionRequested

```

可以看到这里这个位置是 `ijklmnopqrstuvwx`，那么其实AES的密钥就是 `ijklmnopqrstuvwx`

而base64的密钥就是strings中看见的那个字符串，也就是

```

db 'c' ; DATA XREF: sub_40103F+1F1r
db 'DEFGHIJKLMNOPQRSTUVWXYZABcdefghijklmnopqrstuvwxyzab0123456789+/',0
align 4

```

问题6

对于加密算法，它的密钥足够可靠吗？另外你必须知道什么？

在main函数中我们可以看见

```

0040183C push    0                ; dwCreationFlags
0040183E lea     ecx, [ebp+var_58]
00401841 push    ecx                ; lpParameter
00401842 push    offset sub_40132B ; lpStartAddress
00401847 push    0                ; dwStackSize
00401849 push    0                ; lpThreadAttributes
0040184B call    ds:CreateThread
00401851 mov     [ebp+var_20], eax
00401854 cmp     [ebp+var_20], 0
00401858 jnz     short loc_401867

```

这里创建了一个线程，并且线程的起始地址就是加密函数的起始地址

进入到这个函数里，查看一下参数都是在哪里进行了使用

```

00401390 push    0                ; lpOverlapped
00401392 lea     edx, [ebp+NumberOfBytesRead]
00401395 push    edx                ; lpNumberOfBytesRead
00401396 push    400h            ; nNumberOfBytesToRead
0040139B lea     eax, [ebp+Buffer]
004013A1 push    eax                ; lpBuffer
004013A2 mov     ecx, [ebp+var_BE0]
004013A8 mov     edx, [ecx]
004013AA push    edx                ; hFile
004013AB call    ds:ReadFile
004013B1 test    eax, eax
004013B3 jz      short loc_4013BB

```

首先这里有一个readfile的函数，这个函数的参数是var_BE0，

```

mov     eax, [ebp+arg_0]
mov     [ebp+var_BE0], eax

```

而这个BE0来自于arg_0

```

0040183E lea     ecx, [ebp+var_58]
00401841 push    ecx                ; lpParameter
00401842 push    offset sub_40132B ; lpStartAddress

```

也就是创建线程的时候传入的var_58

线程内还有一个writefile的函数，这函数的参数

```

00401449 push    eax                ; lpBuffer
0040144A mov     eax, [ebp+var_BE0]
00401450 mov     ecx, [eax+4]
00401453 push    ecx                ; hFile
00401454 call    ds:WriteFile

```

从边上的注释可以看出来这个参数是arg_10

```

00401587 var_14= dword ptr -14h
00401587 ProcessInformation= _PROCESS_INFORMATION
00401587 arg_10= dword ptr 18h
00401587
00401587 push    ebp

```

往上走可以发现这里其实就是这个函数的一个参数

```

0040196E loc_40196E:
0040196E mov     ecx, [ebp+s]
00401974 push    ecx
00401975 sub     esp, 10h
00401978 mov     edx, esp
0040197A mov     eax, dword ptr [ebp+name.sa_family]
00401980 mov     [edx], eax
00401982 mov     ecx, dword ptr [ebp+name.sa_data+2]
00401988 mov     [edx+4], ecx
0040198B mov     eax, dword ptr [ebp+name.sa_data+6]
00401991 mov     [edx+8], eax
00401994 mov     ecx, dword ptr [ebp+name.sa_data+0Ah]
0040199A mov     [edx+0Ch], ecx
0040199D call    sub_4015B7
004019A2 add     esp, 14h
004019A5 xor     eax, eax

```

从交叉引用可以发现这里的arg_10来自ebp+s

```

00401944 lea     edx, [ebp+name]
0040194A push    edx                ; name
0040194B mov     eax, [ebp+s]
00401951 push    eax                ; s
00401952 call    ds:connect
00401958 mov     [ebp+var_194], eax
0040195E cmp     [ebp+var_194], 0FFFFFFFh
00401965 jnz     short loc_40196E

```

再往上看可以发现这个s就是connect函数创建的socket

回到函数内，我们可以发现

```

9 call ds:CreatePipe
F push 0 ; nSize
1 lea edx, [ebp+PipeAttributes]
4 push edx ; lpPipeAttributes
5 lea eax, [ebp+var_38]
8 push eax ; hWritePipe
9 lea ecx, [ebp+var_5C]
C push ecx ; hReadPipe
D call ds:CreatePipe
3 push 2 ; dwOptions
5 push 1 ; bInheritHandle
7 push 0 ; dwDesiredAccess
9 lea edx, [ebp+TargetHandle]
C push edx ; lpTargetHandle
D call ds:GetCurrentProcess
3 push eax ; hTargetProcessHandle
4 mov eax, [ebp+hSourceHandle]
7 push eax ; hSourceHandle
8 call ds:GetCurrentProcess
E push eax ; hSourceProcessHandle
F call ds:DuplicateHandle
5 test eax, eax
7 jnz short loc_401656

```

```

638 call ds:GetCurrentProcess
63E push eax ; hSourceProcessHandle
63F call ds:DuplicateHandle
645 test eax, eax
647 jnz short loc_401656

```

```

icatehandl ; "DuplicateHandle"

```

```

00401656
00401656 loc_401656: ; dwOptions
00401656 push 2
00401658 push 0 ; bInheritHandle
0040165A push 0 ; dwDesiredAccess
0040165C lea ecx, [ebp+var_18]
0040165F push ecx ; lpTargetHandle
00401660 call ds:GetCurrentProcess
00401666 push eax ; hTargetProcessHandle
00401667 mov edx, [ebp+hObject]
0040166A push edx ; hSourceHandle
0040166B call ds:GetCurrentProcess
00401671 push eax ; hSourceProcessHandle
00401672 call ds:DuplicateHandle
00401678 test eax, eax
0040167A jnz short loc_401689

```

```

00401770 push 0 ; lpInreadAttributes
00401772 push 0 ; lpProcessAttributes
00401774 push offset CommandLine ; "cmd.exe"
00401779 push 0 ; lpApplicationName
0040177B call ds:CreateProcessA
00401781 mov [ebp+var_34], eax
00401784 mov eax, [ebp+ProcessInformation.hProcess]
00401787 mov dword_41336C, eax
0040178C mov ecx, [ebp+hSourceHandle]
0040178F push ecx ; hObject
00401790 call ds:CloseHandle
00401796 test eax, eax
00401798 inz short loc_4017A7

```

这一系列的操作就是典型的创建了一个反向shell，建立后门，使用 CreateProcessA 进行启动。而根据之前的调用base64和AES的位置我们可以发现，这两个都是在readfile和writefile之间被调用的，然后base64的调用是在AES之前，也就是说，这两个应该是有个先后顺序：首先使用base64对传递来的指令进行一个解密操作。然后在本地执行完指令之后，使用AES对执行结果进行加密，并反馈给远端。

问题7

恶意代码做了什么？

根据刚刚的分析可以知道，这个恶意代码还建立了一个shell的后门，并且后门是使用base64对发过来的指令进行解密，用AES对执行结果进行加密再发送回去

问题8

构造代码来解密动态分析过程中生成的一些内容，解密后的内容是什么？

两个加密方式中base64相对较为简单，这里先尝试解密Base64产生的一些内容

假设截取到的网络通信的部分信息为 `BInaEi==`

python脚本如下：

```
1 import string
2 import base64
3
4 result = ""
5 cipher_content =
6 "CDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"
7 standard_b64 =
8 "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"
9
10 cipher_text = "BInaEi=="
11
12 for each_ch in cipher_text:
13     if each_ch in cipher_content:
14         result += standard_b64[string.find(cipher_content, str(each_ch))]
15     elif each_ch == '=':
16         s += '='
17
18 result = base64.decodestring(result)
19 print(result)
```

得到的解密结果为：`dir`，也就是说此时攻击者执行的指令是dir，想要获得当前路径下的目录列表

再尝试使用python解密AES的内容，在wireshark中抓到内容为：

```
00000000 37 f3 1f 04 51 20 e0 b5 86 ac b6 0f 65 20 89 92 7...Q .. ....e ..
00000010 4f af 98 a4 c8 76 98 a6 4d d5 51 8f a5 cb 51 c5 0....v.. M.Q...Q.
00000020 cf 86 11 0d c5 35 38 5c 9c c5 ab 66 78 40 1d df .....58\ ...fx@..
00000030 4a 53 f0 11 0f 57 6d 4f b7 c9 c8 bf 29 79 2f c1 35...WmO ....)y/.
00000040 ec 60 b2 23 00 7b 28 fa 4d c1 7b 81 93 bb ca 9e .`.#.{(. M.{.....
00000050 bb 27 dd 47 b6 be 0b 0f 66 10 95 17 9e d7 c4 8d .'.G.... f.....
00000060 ee 11 09 99 20 49 3b df de be 6e ef 6a 12 db bd .... I;. ..n.j...
00000070 a6 76 b0 22 13 ee a9 38 2d 2f 56 06 78 cb 2f 91 .v."...8 -/V.x./.
```

python脚本为：

```

1 from Crypto.Cipher import AES
2 import binascii
3
4 cipher_text = "37 f3 1f 04 51 20 e0 b5 86 ac b6 b5 20 89 92 4f af 98 a4 c8
76 98 a6 4d d5 51 8f a5 cb 51 c5 cf 86 11 0d c5 35 38 5c 9c c5 ab 66 78 40 1d
df 4a 53 f0 11 0f 57 6d 4f b7 c9 c8 bf 29 79 2f c1 ec 60 b2 23 00 7b 28 fa 4d
c1 7b 81 93 bb ca 9e bb 27 dd 47 b6 be 0b 0f 66 10 95 17 9e d7 c4 8d ee 11 09
99 20 49 3b df de be 6e ef 6a 12 db bd a6 76 b0 22 13 ee a9 38 2d 2f 56 06 78
cb 2f 91 af 64 af a6 d1 43 f1 f5 47 f6 c2 c8 6f 00 49 39"
5
6 cipher_text = binascii.unhexlify(cipher_text.replace(' ', ''))
7 obj = AES.new('ijklmnopqrstuvw', AES.MODE_CBC)
8 print(obj.decrypt(cipher_text))

```

得到的解密结果

Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

Yara

本次的样本都是在进行加解密操作，具有的共性的显式特征比较少，相对来说就比较难以提取特征字符串

本次编写的yara规则如下

```

1 import "pe"
2
3 rule URL {
4     strings:
5         $Http = "http://" nocase
6         $Https = "https://" nocase
7         $www = "www."
8     condition:
9         $Http or $Https or $www
10 }
11
12 rule StandardBase64 {
13     strings:
14         $base =
15         "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"
16     condition:
17         $base
18 }
19
20 rule ShellCmd {
21     strings:
22         $exe = "cmd.exe"
23     condition:
24         $exe
25 }

```

检测结果

```
D:\Study\terms\3. Junior\FirstSemester\计算机病毒与防治技术（王志）\homework>yara64.exe -r yara_rules\lab13.yar Chapter_13L
URL Chapter_13L\Lab13-03.exe
ShellCmd Chapter_13L\Lab13-03.exe
URL Chapter_13L\Lab13-01.exe
StandardBase64 Chapter_13L\Lab13-01.exe
```