

南開大學

## 恶意代码分析与防治课程实验报告

### 实验十三：恶意代码加密技术



学 院 网络空间安全学院  
专 业 信息安全  
学 号 2111033  
姓 名 艾明旭  
班 级 信息安全一班

## 一、实验目的

通过分析恶意代码的加密行为，检测恶意代码通过何种方式隐藏自己的进程，从而选择合适的解密方案，能够有效的将恶意代码解密。

恶意代码编写者与恶意代码分析人员都在不断提高他们的能力和技巧。为了逃避检测与阻挠分析，恶意代码编写者用越来越多的方法来保护他们的目的、技术及通信内容。他们使用的主要工具是编码和加密。编码不仅仅影响通信，它也可以使恶意代码更加难以分析和理解。幸运的是，利用合适的工具，正在使用的很多技术都相对容易识别与对付。

本章涵盖了恶意代码最常使用的加密和解密技术。也讨论识别，了解它们的一些工具和技术，同时还讨论了恶意代码使用的编码和解码方法。

本章侧重于通用的加密算法，并且解释了如何识别它们以及如何执行解密。下一章，我们会专门介绍恶意代码是如何使用网络命令和控制的。在许多情况下，这种网络命令和控制浏览是加密的，但是仍然可能创建可靠的特征探测恶意通信。

数据加密指的是以隐藏恶意代码目的的加密行为，分析加密算法有两步骤：**识别，解密**

## 二、实验原理

### 简单加密算法

特点：使用代码量少

凯撒密码：字符串向右移动字符

异或加密：异或加密只需要一个字符即可，单字节加密可通过暴力破解来解密（暴力破解识别）

- 保留 NULL 的单字节 XOR 加密：遇到 NULL 和密钥本身则不加密，这种更隐蔽
- 分析的时候看到异或在循环里，可能是在加解密

其他简单算法：

- ADD 和 SUB 组合：一个加，一个减，配套使用
- ROL, ROR：旋转字节左右的比特位
- ROT：凯撒密码，通常使用可打印字符
- Multibyte：按块（4 字节或 8 字节）进行异或
- 链或环：使用内容本身作为密钥，最常见的是原始密钥在明文两端，编码过的输出字符作为下一个字符的密钥

Base64：比较常见，广泛用于 HTTP 和 XML

- 长度有限的随机字符，如果被补齐，长度通常可以被 4 整除
- 通常填充字符是 `=`
- Base64 加密最精彩的地方是可以自定义加密索引

### 常见加密算法

可以通过识别字符串和导入来判断使用了什么加密库

通过识别加密常量也可以获取加密相关信息，使用 IDA 的插件可进行操作

另一种识别方法是通过查找高熵值内容

通过组合加密手段来进行自定义加密

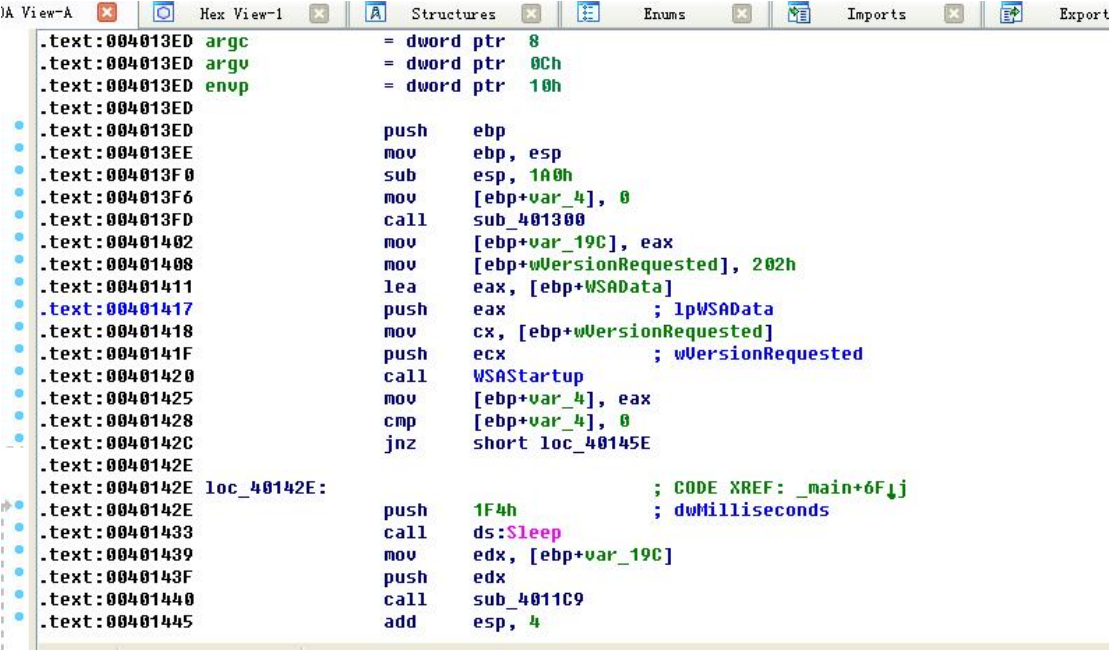
### 常见解密方法

最经济的解密方法是把程序运行起来让其自解密

也可以手动执行解密函数，自己编写解密函数或者调用程序自己的解密函数

## 三、实验过程

首先打开 lab13-01.exe



```
.text:004013ED  argc      = dword ptr 8
.text:004013ED  argv      = dword ptr 0Ch
.text:004013ED  envp      = dword ptr 10h
.text:004013ED
.text:004013ED  push     ebp
.text:004013EE  mov      ebp, esp
.text:004013F0  sub      esp, 1A0h
.text:004013F6  mov      [ebp+var_4], 0
.text:004013FD  call     sub_401300
.text:00401402  mov      [ebp+var_19C], eax
.text:00401408  mov      [ebp+wVersionRequested], 202h
.text:00401411  lea      eax, [ebp+WSAData]
.text:00401417  push     eax ; lpWSAData
.text:00401418  mov      cx, [ebp+wVersionRequested]
.text:0040141F  push     ecx ; wVersionRequested
.text:00401420  call     WSASStartup
.text:00401425  mov      [ebp+var_4], eax
.text:00401428  cmp      [ebp+var_4], 0
.text:0040142C  jnz      short loc_40145E
.text:0040142E
.text:0040142E  loc_40142E: ; CODE XREF: _main+6F↓j
.text:0040142E  push     1F4h ; dwMilliseconds
.text:00401433  call     ds:Sleep
.text:00401439  mov      edx, [ebp+var_19C]
.text:0040143F  push     edx
.text:00401440  call     sub_4011C9
.text:00401445  add      esp, 4
```

### 问题 1

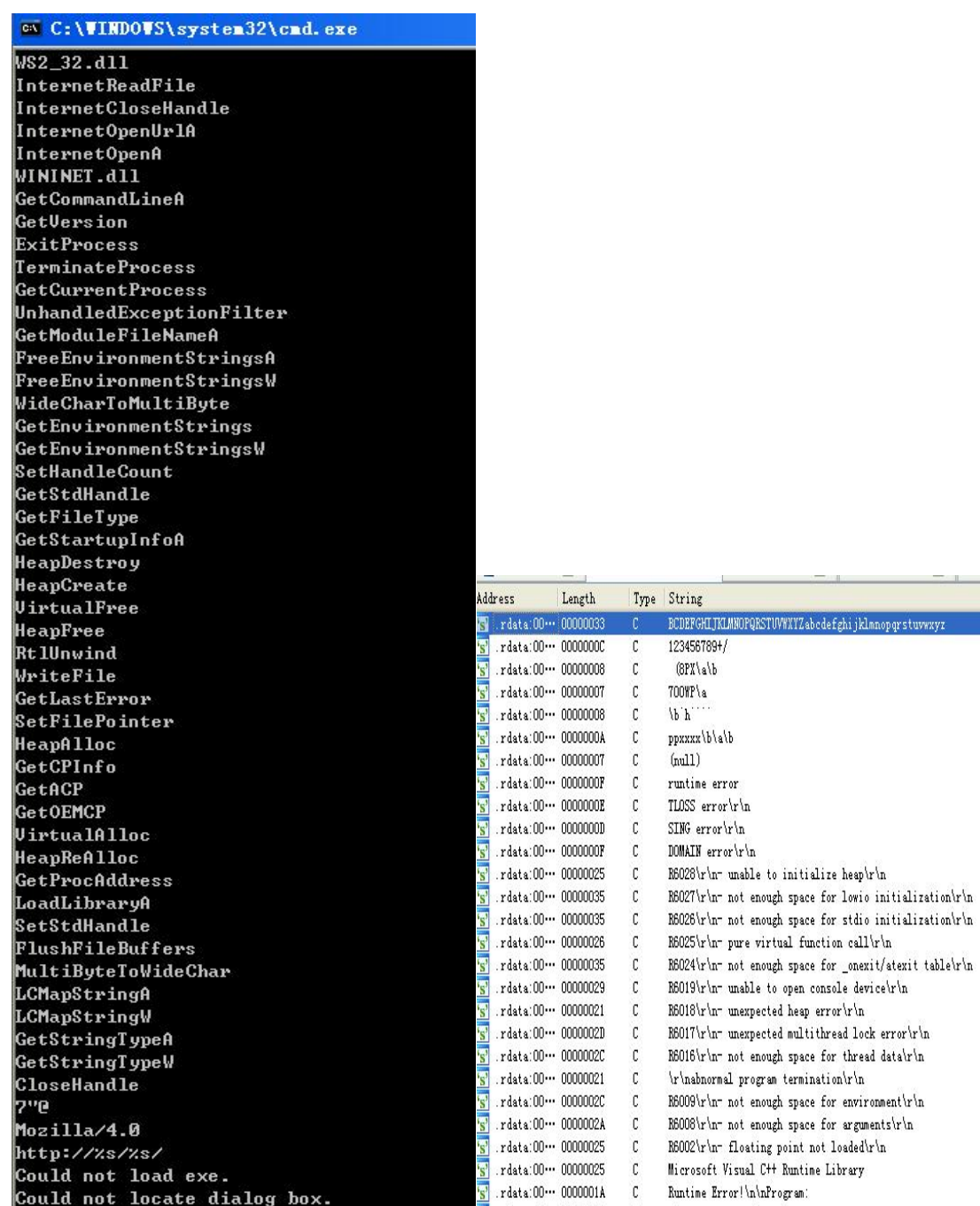
比较恶意代码中的字符串（字符串命令的输出）与动态分析提供的有用信息，基于这些比较，哪些元素可能被加密？

答： 动态分析出的网络行为中出现了两个 strings 工具未检测出的字符串，分别是 eHBsaQ==和 [www.practicalmalwareanalysis.com](http://www.practicalmalwareanalysis.com)，可能被加密了。

分析过程：

网络中出现两个恶意代码中不存在的字符串(当 strings 命令运行时,并没有字符串输出)。一个字符串是域名`www.practicalmalwareanalysis.com`， 另外一个一个是 GET 请求路径，它看起来像`aG9zdG5hbWUtZm9v`。加密的是要访问的恶意网站，在 main 函数中，最上面进行了字符串解密，然后在网络功能初始化之后使用该字符串作为参数进行操作，应该是网址。

使用 strings 简单查看一下这个程序



```
C:\WINDOWS\system32\cmd.exe

WS2_32.dll
InternetReadFile
InternetCloseHandle
InternetOpenUrlA
InternetOpenA
MININET.dll
GetCommandLineA
GetVersion
ExitProcess
TerminateProcess
GetCurrentProcess
UnhandledExceptionFilter
GetModuleFileNameA
FreeEnvironmentStringsA
FreeEnvironmentStringsW
WideCharToMultiByte
GetEnvironmentStrings
GetEnvironmentStringsW
SetHandleCount
GetStdHandle
GetFileType
GetStartupInfoA
HeapDestroy
HeapCreate
VirtualFree
HeapFree
RtlUnwind
WriteFile
GetLastError
SetFilePointer
HeapAlloc
GetCPIInfo
GetACP
GetOEMCP
VirtualAlloc
HeapReAlloc
GetProcAddress
LoadLibraryA
SetStdHandle
FlushFileBuffers
MultiByteToWideChar
LCMapStringA
LCMapStringW
GetStringTypeA
GetStringTypeW
CloseHandle
?
Mozilla/4.0
http://%s/%s/
Could not load exe.
Could not locate dialog box.
```

Address	Length	Type	String
.rdata:00000000	00000033	C	BCDEF GHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
.rdata:00000000	0000000C	C	123456789+/-
.rdata:00000000	00000008	C	(8PX\ab
.rdata:00000000	00000007	C	700WF\ab
.rdata:00000000	00000008	C	\b\
.rdata:00000000	0000000A	C	ppxxxx\b\ab
.rdata:00000000	00000007	C	(null)
.rdata:00000000	0000000F	C	runtime error
.rdata:00000000	0000000E	C	TLOSS error\r\n
.rdata:00000000	0000000D	C	SING error\r\n
.rdata:00000000	0000000F	C	DOMAIN error\r\n
.rdata:00000000	00000025	C	B6028\r\n- unable to initialize heap\r\n
.rdata:00000000	00000035	C	B6027\r\n- not enough space for lowio initialization\r\n
.rdata:00000000	00000035	C	B6026\r\n- not enough space for stdio initialization\r\n
.rdata:00000000	00000026	C	B6025\r\n- pure virtual function call\r\n
.rdata:00000000	00000035	C	B6024\r\n- not enough space for _onexit/_atexit table\r\n
.rdata:00000000	00000029	C	B6019\r\n- unable to open console device\r\n
.rdata:00000000	00000021	C	B6018\r\n- unexpected heap error\r\n
.rdata:00000000	0000002D	C	B6017\r\n- unexpected multithread lock error\r\n
.rdata:00000000	0000002C	C	B6016\r\n- not enough space for thread data\r\n
.rdata:00000000	00000021	C	\r\nabnormal program termination\r\n
.rdata:00000000	0000002C	C	B6009\r\n- not enough space for environment\r\n
.rdata:00000000	0000002A	C	B6008\r\n- not enough space for arguments\r\n
.rdata:00000000	00000025	C	B6002\r\n- floating point not loaded\r\n
.rdata:00000000	00000025	C	Microsoft Visual C++ Runtime Library
.rdata:00000000	0000001A	C	Runtime Error!\n\nProgram:

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： KIZXORXZWVZWLZI^ZUZWBHRH ，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	6.48260800	192.0.78.25	192.168.159.131	HTTP	461	HTTP/1.1 301 Moved Permanently (text/html)

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。与找到的字符串进行对比，相同的有 Mozilla/4.0，却没有看到 eHBsaQ== 和 www.practicalmalwareanalysis.com，推测这些字符串被加密了。

## 问题 2

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

答：只找到了一处，004011B8 处的 xor eax, 38h 是一个 XOR 循环加密的指令。

分析过程：

使用 IDA 打开程序，并进行搜索

The screenshot shows the IDA Pro interface with assembly code loaded. A 'Text search (slow!)' dialog box is open, displaying the search results for the string 'xor'. The search parameters are set to 'Case sensitive', 'Regular expression', and 'Identifier'. The search direction is set to 'Search Down'. The search results show the string 'xor' found at address 004011B8, which corresponds to the instruction 'xor eax, 38h' in the assembly code.

```

.text:0040145E loc_40145E: ; CODE XREF: _main+3F↑j
.text:0040145E call    WSACleanup
.text:00401463 xor     eax, eax
.text:00401465 mov     esp, ebp
.text:00401467 pop     ebp
.text:00401468 retn
.text:00401468 _main    endp
.text:00401468 ; -----
.text:00401469 align 2
.text:0040146A ; [00000006 BYTES: COLLAPSED FUNCTION gethostname. PRESS KEYPAD CTRL-"" TO EX
.text:00401470 ; [00000006 BYTES: COLLAPSED FUNCTION WSACleanup. PRESS KEYPAD CTRL-"" TO EXF
.text:00401476 ; [00000006 BYTES: COLLAPSED FUNCTION WSStartup. PRESS KEYPAD CTRL-"" TO EXF
.text:0040147C align 10h
.text:00401480 ; [0000007B BYTES: COLLAPSED FUNCTION _strlen. PRESS KEYPAD CTRL-"" TO EXPAN
.text:004014FB align 10h
.text:00401500 ; [000000FE BYTES: COLLAPSED FUNCTION _strncpy. PRESS KEYPAD CTRL-"" TO EXPAN
.text:004015FE ; [00000052 BYTES: COLLAPSED FUNCTION sprintf. PRESS KEYPAD CTRL-"" TO EXPAN
.text:0040145C jz      short loc_40142E
.text:0040145E loc_40145E: ; CODE XREF: _main+
.text:0040145E call    WSACleanup
.text:00401463 xor     eax, eax
.text:00401465 mov     esp, ebp
.text:00401467 pop     ebp

```

Text search (slow!)

String: xor

Parameters:

- ☐ Case sensitive
- ☐ Regular expression
- ☐ Identifier
- ☐ Find all occurrences

Direction:

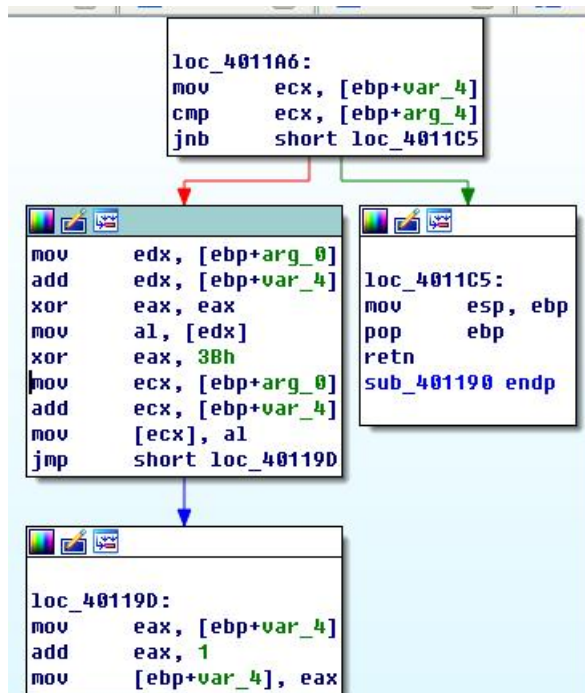
- ☒ Search Down
- ☐ Search Up

OK Cancel



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。



这里就是一个循环进行异或的操作，它递增 var\_4，并且用 0x3B 与 edx 中的内容进行异或，edx 中存的是缓冲区[ebp+arg\_0]偏移[ebp+var\_4]的值，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。

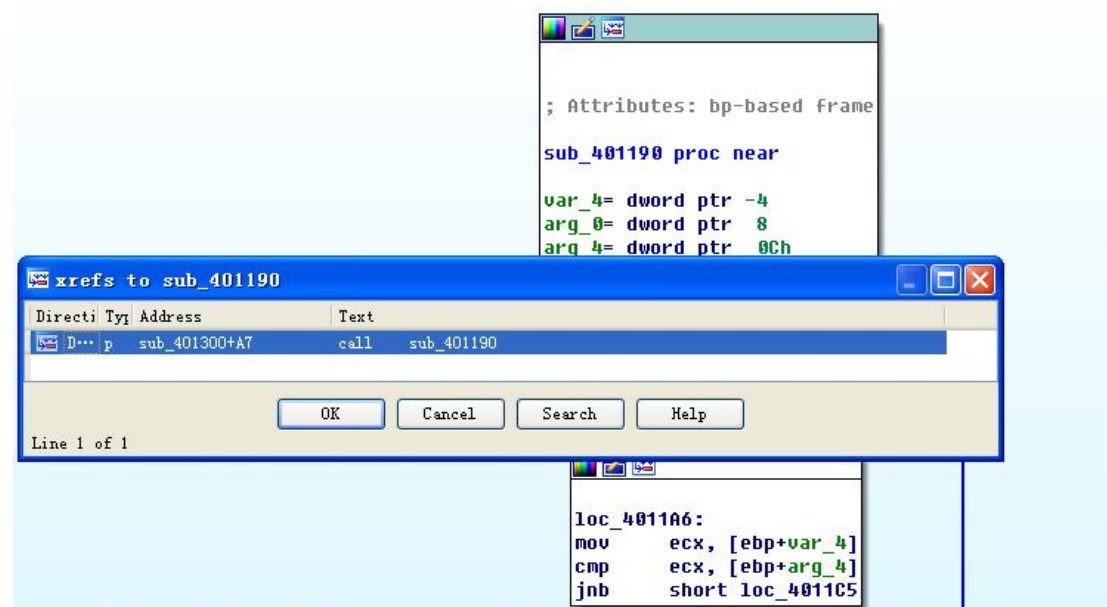
问题 3

恶意代码使用什么密钥加密，加密了什么内容？

答：密钥是 0x3B，加密了域名 [www.practicalmalwareanalysis.com](http://www.practicalmalwareanalysis.com)。

分析过程：

根据刚刚的分析可以知道这个异或加密的密钥是 3Bh



通过查找交叉引用，发现该加密函数仅被函数 sub\_401300 调用。于是查看函数 sub\_401300。可以看到在调用 加密函数之前有一系列对资源节的操作：依次调用了 GetModuleHandleA、FindResourceA、SizeofResource、GlobalAlloc、LoadResource 和 LockResource，通过 FindResource 的参数来寻找操作的 对象：其中 lpType=0xA，表示资源数据原始数据，lpName=65h，在这里它是一个索引号，表示引用 ID 为 0x65 的资源。

通过查找交叉引用，发现该加密函数仅被函数 sub\_401300 调用。于是查看函数 sub\_401300。可以看到在调用 加密函数之前有一系列对资源节的操作：依次调用了 GetModuleHandleA、FindResourceA、SizeofResource、GlobalAlloc、LoadResource 和 LockResource，通过 FindResource 的参数来寻找操作的 对象：其中 lpType=0xA，表示资源数据原始数据，lpName=65h，在这里它是一个索引号，表示引用 ID 为 0x65 的资源。

```

loc_401392:
mov     eax, [ebp+hResData]
push    eax                ; hResData
call    ds:LockResource
mov     [ebp+var_10], eax
mov     ecx, [ebp+dwBytes]
push    ecx
mov     edx, [ebp+var_10]
push    edx
call    sub_401190
add     esp, 8
mov     eax, [ebp+var_10]
jmp     short loc_4013E9

var_4= dword ptr -4

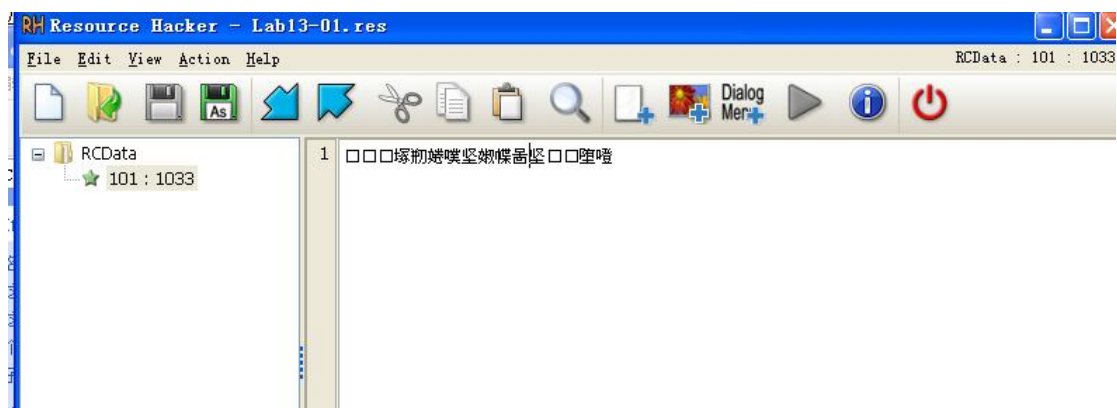
push    ebp
mov     ebp, esp
sub     esp, 28h
mov     [ebp+var_24], 0
mov     [ebp+var_10], 0
push    0                  ; lpModuleName
call    ds:GetModuleHandleA
mov     [ebp+hModule], eax
cmp     [ebp+hModule], 0
jnz     short loc_401339

loc_401357:
mov     ecx, [ebp+hResInfo]
push    ecx                ; hResInfo
mov     edx, [ebp+hModule]
push    edx                ; hModule
call    ds:SizeofResource
mov     [ebp+dwBytes], eax
mov     eax, [ebp+dwBytes]
push    eax                ; dwBytes
push    40h                ; uFlags
call    ds:GlobalAlloc
mov     [ebp+var_4], eax
mov     ecx, [ebp+hResInfo]
push    ecx                ; hResInfo
mov     edx, [ebp+hModule]
push    edx                ; hModule
call    ds:LoadResource
mov     [ebp+hResData], eax
cmp     [ebp+hResData], 0
jnz     short loc_401392

loc_401339:
; lpType
push    0Ah
push    65h                ; lpName
mov     eax, [ebp+hModule]
push    eax                ; hModule
call    ds:FindResourceA
mov     [ebp+hResInfo], eax
cmp     [ebp+hResInfo], 0
jnz     short loc_401357

```

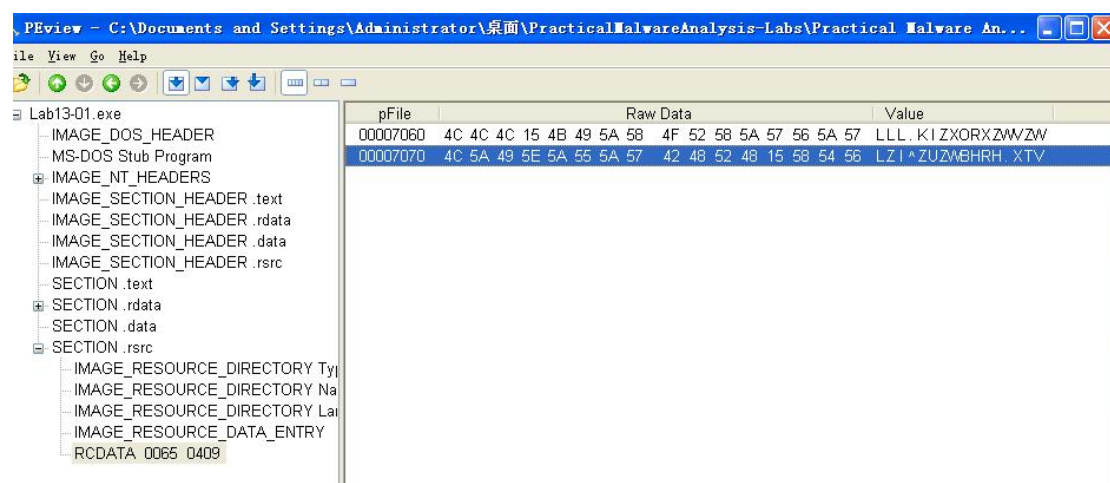
可以看见在异或操作之前，这里释放了资源节的内容，那么很明显这个异或操作就是对资源节中的内容进行解密。



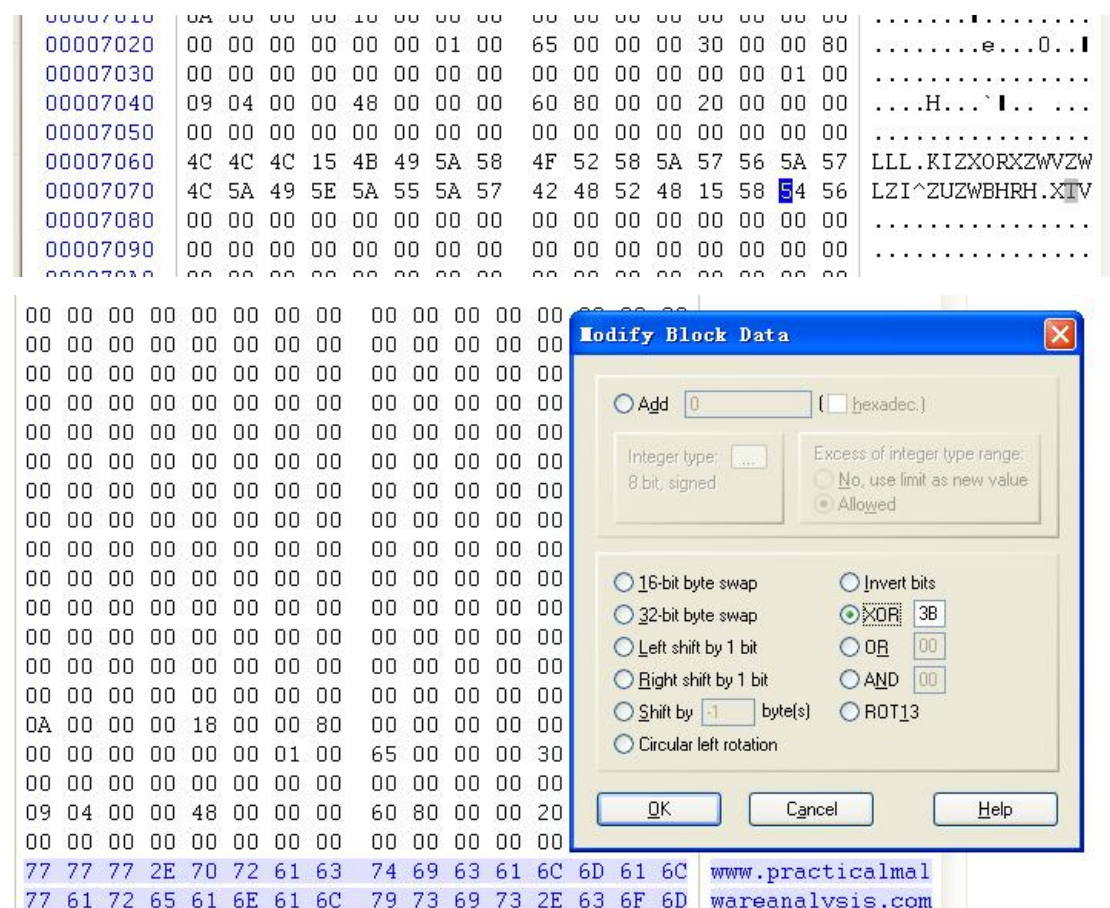
于是我们使用 PView 查看恶意代码的资源节，查找索引号为 65h 的资源，偏移为 0x7060，然后用 WinHex 打开，定位到该资源节处，选择 Edit->Modify Data->XOR，输入 3B，得到异或结果，为域名 [www.practicalmalwareanalysis.com](http://www.practicalmalwareanalysis.com)。

同时通过 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。





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：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 熵插件识别一些其他类型的加密机制，你发现了什么？

答：用 PEiD 的 KANAL 插件和 IDA 的熵插件，发现了恶意代码使用的 Base64 编码字符串：

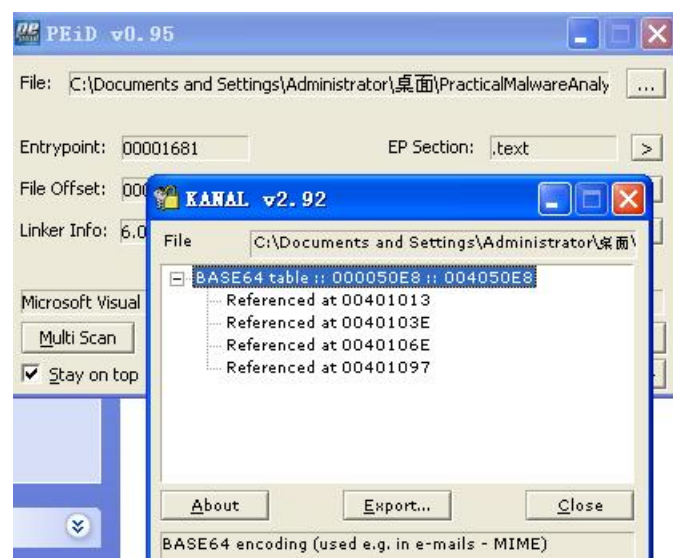
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/

分析过程：

使用 PEiD 的 Kryo ANALyzer 插件,发现该恶意代码有一个 BASE64 table,地址是 0x004050E8,

于是我们在 IDA 中跳转到该位置,看到这里存着的是字符串

ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/-。



IDA 熵插件也能识别一个自定义的 Base64 索引字符串,这表明没有明显的证据与 xor 指令

相关。我们知道了 s\_xor2,s\_xor4 和 AES 加密相关,而 s\_xor3,x\_xor5 与 AES 解密相关

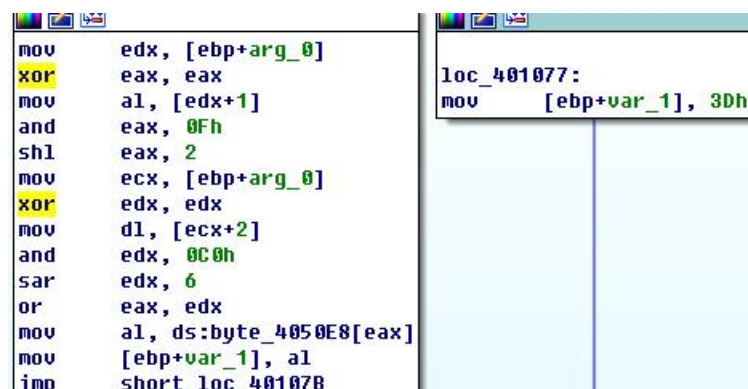
```
.rdata:004050E4          align 8
.rdata:004050E8  byte_4050E8  db 41h          ; DATA XREF: sub_401000+11↑r
.rdata:004050E8                                     ; sub_401000+3C↑r ...
.rdata:004050E9          db 42h ; B
.rdata:004050EA          db 43h ; C
.rdata:004050EB          db 44h ; D
.rdata:004050EC          db 45h ; E
.rdata:004050ED          db 46h ; F
.rdata:004050EE          db 47h ; G
.rdata:004050EF          db 48h ; H
.rdata:004050F0          db 49h ; I
.rdata:004050F1          db 4Ah ; J
.rdata:004050F2          db 4Bh ; K
.rdata:004050F3          db 4Ch ; L
.rdata:004050F4          db 4Dh ; M
.rdata:004050F5          db 4Eh ; N
.rdata:004050F6          db 4Fh ; O
.rdata:004050F7          db 50h ; P
.rdata:004050F8          db 51h ; Q
.rdata:004050F9          db 52h ; R
.rdata:004050FA          db 53h ; S
.rdata:004050FB          db 54h ; T
.rdata:004050FC          db 55h ; U
.rdata:004050FD          db 56h ; V
.rdata:004050FE          db 57h ; W
```

这个字符串在函数 sub\_401000 中被引用了四次,注意到该函数中有一处对=的引用,这是

Base64 加密中的填充 字符,因此该函数执行加密操作,调用该函数的函数是 Base64 加密

函数。可以看到,sub\_4010B1 调用了该函 数,于是进入这个函数查看。调用了 strlen 函数

得到参数中字符串的长度，将该字符串以 3 字节为单位分组， 分别传给 sub\_401000 进行加密操作，得到该分组的 4 字节密文。



函数 sub\_4011C9 调用了这个加密函数，查看其参数，源字符串是 strncpy 的返回值，而 strncpy 的输入是函数 gethostname 的输出，即要加密的字符串是主机名的前 12 个字节。而其目的字符串作为 sprintf 的参数，即加密后的字符串被输出，也就是我们在动态分析阶段看到的那样。

```

push    ebp
mov     ebp, esp
sub     esp, 558h
mov     byte ptr [ebp+var_30], 0
xor     eax, eax
mov     [ebp+var_30+1], eax
mov     [ebp+var_2B], eax
mov     [ebp+var_27], eax
mov     [ebp+var_23], eax
push    offset aMozilla4_0 ; "Mozilla/4.0"
lea     ecx, [ebp+szAgent]
push    ecx                ; char *
call    _sprintf
add     esp, 8
push    100h                ; namelen
lea     edx, [ebp+name]
push    edx                ; name
call    gethostname
mov     [ebp+var_4], eax
push    0Ch                ; size_t
lea     eax, [ebp+name]
push    eax                ; char *
lea     ecx, [ebp+var_18]
push    ecx                ; char *

```

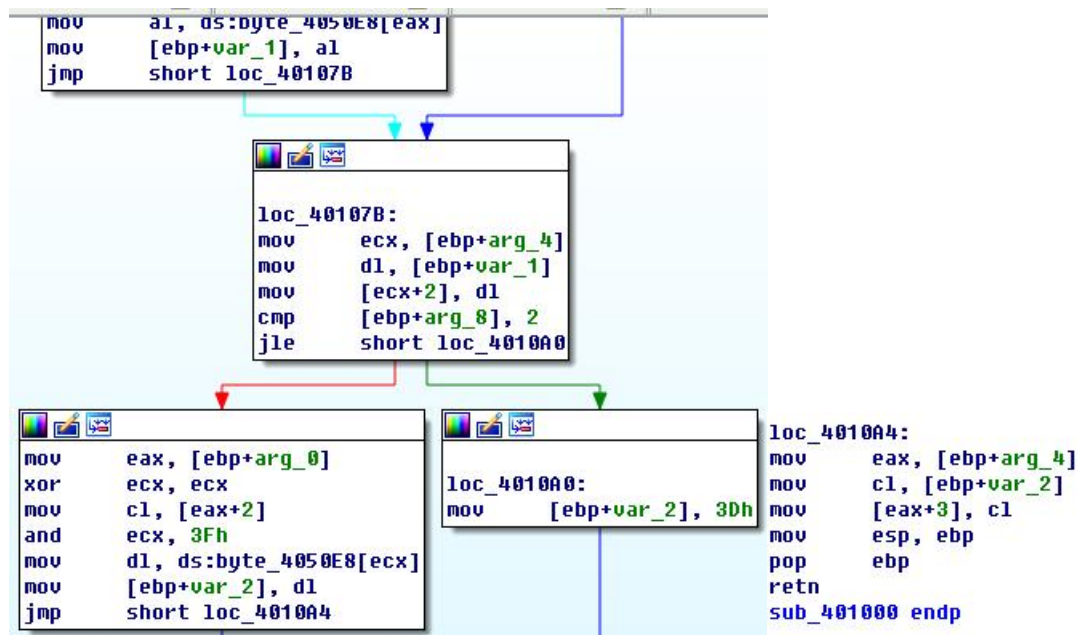
## 问题 5

什么类型的加密被恶意代码用来发送部分网络流量？

答： 使用 Base64 加密来构造 GET 请求字符串。

分析过程：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

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？

答： 从 0x004010B1 处开始。

; Attributes: bp-based frame

base64 proc near

```

var_2= byte ptr -2
var_1= byte ptr -1
arg_0= dword ptr 8
arg_4= dword ptr 0Ch
arg_8= dword ptr 10h

```

```

push
mov
push
mov
xor
mov
mov
sar
mov

```



分析过程：

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 base64，再次通过交叉应用可以发现

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？

答： 12 字节，加密了主机名，这使得用于构造 GET 请求的字符串不超过 16 字节。

分析过程：

使用 F5 键进入到 C 语言代码的形式



```

int __cdecl base64(int a1, int a2, signed int a3)
{
    int result; // eax@7
    char v4; // [sp+0h] [bp-2h]@5
    char v5; // [sp+1h] [bp-1h]@2

    *(_BYTE *)a2 = byte_4050E8[(signed int)*(_BYTE *)a1 >> 2];
    *(_BYTE *)(a2 + 1) = byte_4050E8[((*(_BYTE *)a1 + 1) & 0xF0) >> 4 | 16 * (*(_BYTE *)a1 & 3)];
    if ( a3 <= 1 )
        v5 = 61;
    else
        v5 = byte_4050E8[((*(_BYTE *)a1 + 2) & 0xC0) >> 6 | 4 * (*(_BYTE *)a1 + 1) & 0xF];
    *(_BYTE *)(a2 + 2) = v5;
    if ( a3 <= 2 )
        v4 = 61;
    else
        v4 = byte_4050E8[*(_BYTE *)a1 + 2) & 0x3F];
    result = a2;
    *(_BYTE *)(a2 + 3) = v4;
    return result;
}

```

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。

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。

```

mov     [ebp+var_27], eax
mov     [ebp+var_23], eax
push    offset aMozilla4_0 ; "Mozilla/4.0"
lea     ecx, [ebp+szAgent]
push    ecx                ; char *
call    _sprintf
add     esp, 8
push    100h               ; namelen
lea     edx, [ebp+name]
push    edx                ; name
call    gethostname
mov     [ebp+var_4], eax
push    0Ch                ; size_t
lea     eax, [ebp+name]
push    eax                ; char *
lea     ecx, [ebp+var_18]
push    ecx                ; char *
call    _strncpy
add     esp, 0Ch
mov     [ebp+var_C], 0
lea     edx, [ebp+var_30]
push    edx                ; int
lea     eax, [ebp+var_18]
push    eax                ; char *
call    sub_4010B1
add     esp, 8
mov     byte ptr [ebp+var_23+3], 0

```

```

loc_4012A6:
lea     edx, [ebp+dwNumberOfBytesRead]
push    edx                ; lpdwNumberOfBytesRead
push    200h               ; dwNumberOfBytesToRead
lea     eax, [ebp+Buffer]
push    eax                ; lpBuffer
mov     ecx, [ebp+hFile]
push    ecx                ; hFile
call    ds:InternetReadFile
mov     [ebp+var_8], eax
cmp     [ebp+var_8], 0
jnz     short loc_4012EA

loc_4012EA:
movsx   ecx, [ebp+Buffer]
cmp     ecx, 6Fh
jnz     short loc_4012FA

```

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

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==）？

答： 本次得到的加密数据是 eHBsaQ==，有填充字符，这是因为本机的主机名小于 12 字节并且不能被 3 整除。

分析过程：

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？

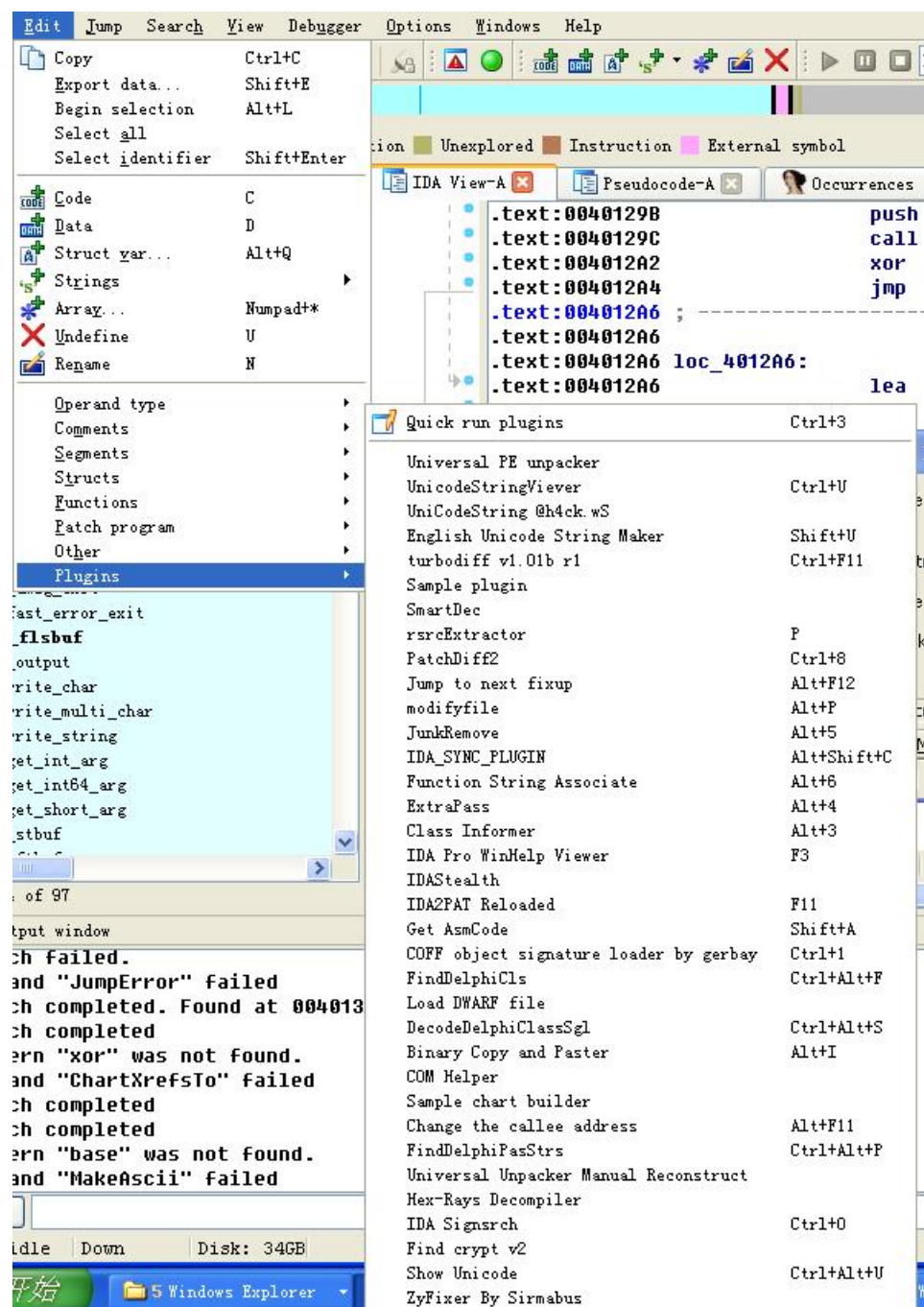
答： 这个恶意代码用经 Base64 加密的主机名构造出一个 GET 发出请求，当接收到以 o 开始的 Web 响应后才会终止。

```
text:00401298      push     ecx                ; hInternet
text:0040129C      call    ds:InternetCloseHandle
text:004012A2      xor     al, al
text:004012A4      jmp     short loc_4012FC
text:004012A6      ; -----|-----
text:004012A6      loc_4012A6:                ; CODE XREF: sub_4011C9+CA↑
text:004012A6      lea     edx, [ebp+dwNumberOfBytesRead]
text:004012A9      push    edx                ; lpdwNumberOfBytesRead
text:004012AA      push    200h               ; dwNumberOfBytesToRead
text:004012AF      lea     eax, [ebp+Buffer]
text:004012B5      push    eax                ; lpBuffer
text:004012B6      mov     ecx, [ebp+hFile]
text:004012BC      push    ecx                ; hFile
text:004012BD      call    ds:InternetReadFile
text:004012C3      mov     [ebp+var_8], eax
text:004012C6      cmp     [ebp+var_8], 0
text:004012CA      jnz     short loc_4012EA
text:004012CC      mov     edx, [ebp+hInternet]
text:004012D2      push    edx                ; hInternet
text:004012D3      call    ds:InternetCloseHandle
text:004012D9      mov     eax, [ebp+hFile]
text:004012DF      push    eax                ; hInternet
text:004012E0      call    ds:InternetCloseHandle
text:004012E6      xor     al, al
text:004012E8      jmp     short loc_4012FC
```

分析过程：

依次调用了 InternetOpenA、InternetOpenUrlA 和 InternetReadFile，用于打开并且读取之前构成的 URL。并将读到的数据中的第一个字符与字母 o 比较，如果相同则返回 1，否则返回 0。

之后使用 IDA 中的 FindCrype 插件查看

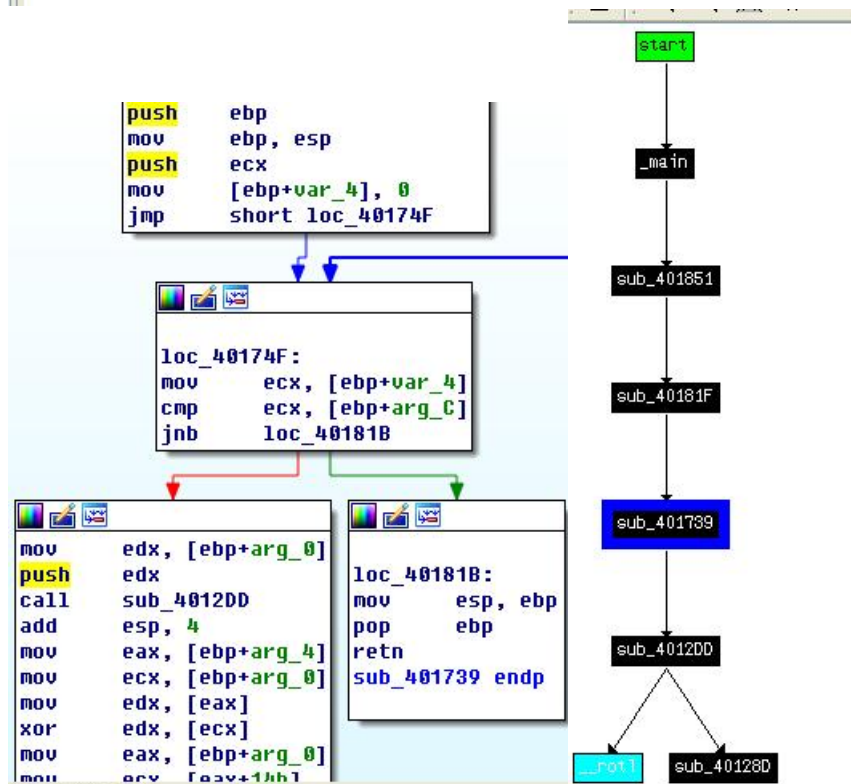


输出框依旧没有什么内容，也就是说这里没有找到东西

接下来搜索 xor 指令。

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 忽略其中用于将寄存器清零的和库函数中的操作，其余的 xor 指令集中在函数 sub\_40128D 和 sub\_401739 中，并且搜索结果中的第三条指令并没有对应的函数。我们先查看拥有最多 xor 指令的函数 sub\_401739，它包含一个循环，循环体中大多是 xor、SHL 和 SHR 指令，只有一次函数调用，调用的是 sub\_4012DD，而这个函数又会调用 sub\_40128D，即拥有 xor 指令的另一个函数，推测这两处 xor 指令相关。

Address	Function	Instruction
.text:00401040	sub_401000	xor eax, eax
.text:004012D6	sub_40128D	xor eax, [ebp+var_10]
.text:0040171F		xor eax, [esi+edx*4]
.text:0040176F	sub_401739	xor edx, [ecx]
.text:0040177A	sub_401739	xor edx, ecx
.text:00401785	sub_401739	xor edx, ecx
.text:00401795	sub_401739	xor eax, [edx+8]
.text:004017A1	sub_401739	xor eax, edx
.text:004017AC	sub_401739	xor eax, edx
.text:004017BD	sub_401739	xor ecx, [eax+10h]
.text:004017C9	sub_401739	xor ecx, eax
.text:004017D4	sub_401739	xor ecx, eax
.text:004017E5	sub_401739	xor edx, [ecx+18h]
.text:004017F1	sub_401739	xor edx, ecx
.text:004017FC	sub_401739	xor edx, ecx
.text:0040191E	_main	xor eax, eax
.text:0040311A		xor dh, [eax]
.text:0040311E		xor [eax], dh
.text:00403688		xor ecx, ecx
.text:004036A5		xor edx, edx



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

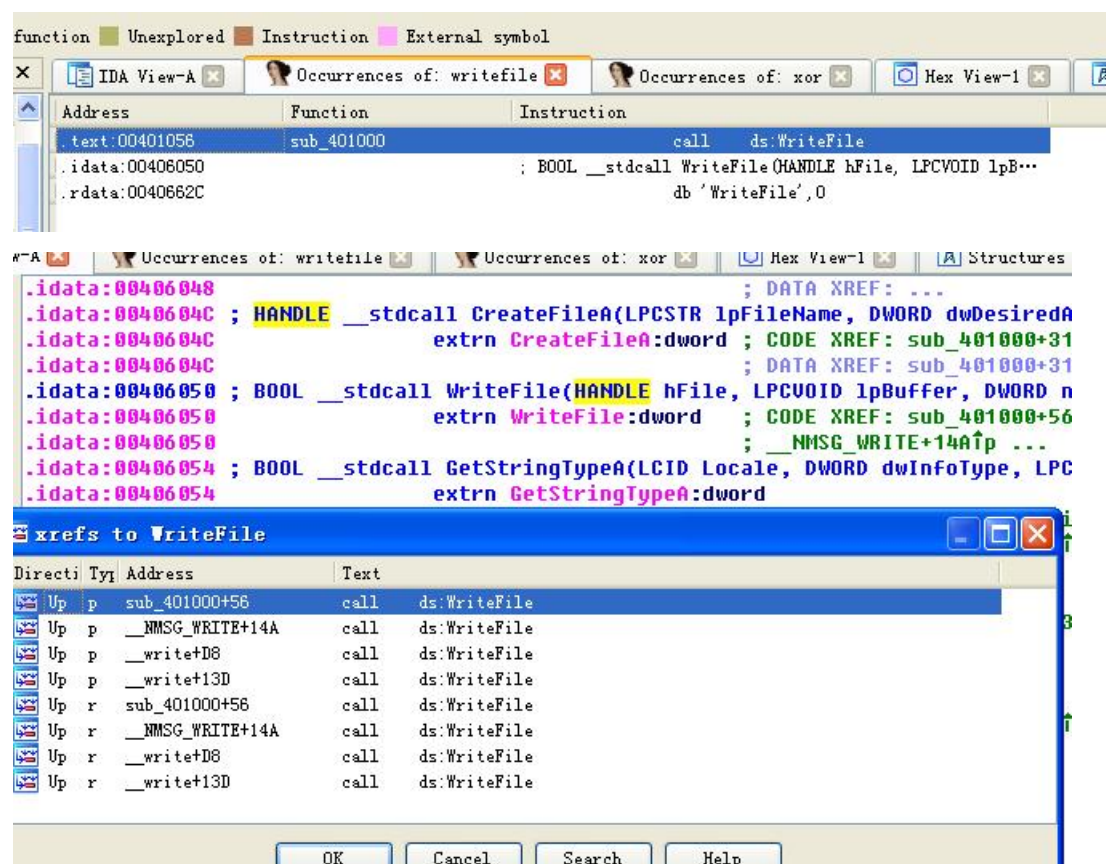
### 问题 3

基于问题 1 的回答，哪些导入函数将是寻找加密函数比较好的一个证据？

答：查看导入函数表，有 WriteFile 函数，由于该恶意代码的行为是不断创建文件并写文件，因此可能会在 WriteFile 函数附近寻找到加密函数。

分析过程：





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？

答：0x0040181F 处。

分析过程：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



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

```
.text:00401885      add     esp, 8
.text:00401888      call    ds:GetTickCount
.text:0040188E      mov     [ebp+var_4], eax
.text:00401891      mov     ecx, [ebp+var_4]
.text:00401894      push    ecx
.text:00401895      push    offset aTemp08x ; "temp%08x"
.text:0040189A      lea     edx, [ebp+FileName]
.text:004018A0      push    edx ; char *
.text:004018A1      call    _sprintf
.text:004018A6      add     esp, 0Ch
.text:004018A9      lea     eax, [ebp+FileName]
.text:004018AF      push    eax ; lpFileName
.text:004018B0      mov     ecx, [ebp+nNumberOfBytesToWrite]
.text:004018B3      push    ecx ; nNumberOfBytesToWrite
.text:004018B4      mov     edx, [ebp+hMem]
.text:004018B7      push    edx ; lpBuffer
.text:004018B8      call    sub_401000
.text:004018BD      add     esp, 0Ch
.text:004018C0      mov     eax, [ebp+hMem]
.text:004018C3      push    eax ; hMem
.text:004018C4      call    ds:GlobalUnlock
.text:004018CA      mov     ecx, [ebp+hMem]
.text:004018CD      push    ecx ; hMem
.text:004018CE      call    ds:GlobalFree
.text:004018D4      mov     esp, ebp
```

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了的时间，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：

```
text:00401851 hMem = dword ptr -0Ch
text:00401851 nNumberOfBytesToWrite = dword ptr -8
text:00401851 var_4 = dword ptr -4
text:00401851
text:00401851      push    ebp
text:00401852      mov     ebp, esp
text:00401854      sub     esp, 20Ch
text:0040185A      mov     [ebp+hMem], 0
text:00401861      mov     [ebp+nNumberOfBytesToWrite], 0
text:00401868      lea     eax, [ebp+nNumberOfBytesToWrite]
text:0040186B      push    eax
text:0040186C      lea     ecx, [ebp+hMem]
text:0040186F      push    ecx
text:00401870      call    sub_401070
text:00401875      add     esp, 8
text:00401878      mov     edx, [ebp+nNumberOfBytesToWrite]
text:0040187B      push    edx
text:0040187C      mov     eax, [ebp+hMem]
text:0040187F      push    eax
text:00401880      call    sub_40181F
text:00401885      add     esp, 8
text:00401888      call    ds:GetTickCount
text:0040188E      mov     [ebp+var_4], eax
text:00401891      mov     ecx, [ebp+var_4]
text:00401894      push    ecx
text:00401895      push    offset aTemp08x ; "temp%08x"
text:0040189A      lea     edx, [ebp+FileName]
```

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

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？

答：当前的屏幕截图。

分析过程：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

```
text:00401076      mov     [ebp+hdc], 0
text:0040107D      push    0                ; nIndex
text:0040107F      call   ds:GetSystemMetrics
text:00401085      mov     [ebp+var_1C], eax
text:00401088      push    1                ; nIndex
text:0040108A      call   ds:GetSystemMetrics
text:00401090      mov     [ebp+cy], eax
text:00401093      call   ds:GetDesktopWindow
text:00401099      mov     hWnd, eax
text:0040109E      mov     eax, hWnd
text:004010A3      push    eax                ; hWnd
text:004010A4      call   ds:GetDC
text:004010AA      mov     hDC, eax
text:004010AF      mov     ecx, hDC
text:004010B5      push    ecx                ; hdc
text:004010B6      call   ds:CreateCompatibleDC
text:004010BC      mov     [ebp+hdc], eax
text:004010BF      mov     edx, [ebp+cy]
text:004010C2      push    edx                ; cy
text:004010C3      mov     eax, [ebp+var_1C]
text:004010C6      push    eax                ; cx
text:004010C7      mov     ecx, hDC
text:004010CD      push    ecx                ; hdc
text:004010CE      call   ds:CreateCompatibleBitmap
text:004010D4      mov     [ebp+h], eax
text:004010D7      mov     edx, [ebp+h]
```

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

```
t:004011B3      push    eax                ; cLines
t:004011B4      push    0                ; start
t:004011B6      mov     ecx, [ebp+h]
t:004011B9      push    ecx                ; hbm
t:004011BA      mov     edx, hDC
t:004011C0      push    edx                ; hdc
t:004011C1      call   ds:GetDIBits
t:004011C7      mov     eax, [ebp+dwBytes]
t:004011CA      add     eax, 36h
t: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？

答：查看用于加密操作的代码块，发现使用的是用户自定义的算法，而不是什么标准的加密算法。可以通过 Ollydbg 来解密。

分析过程：

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

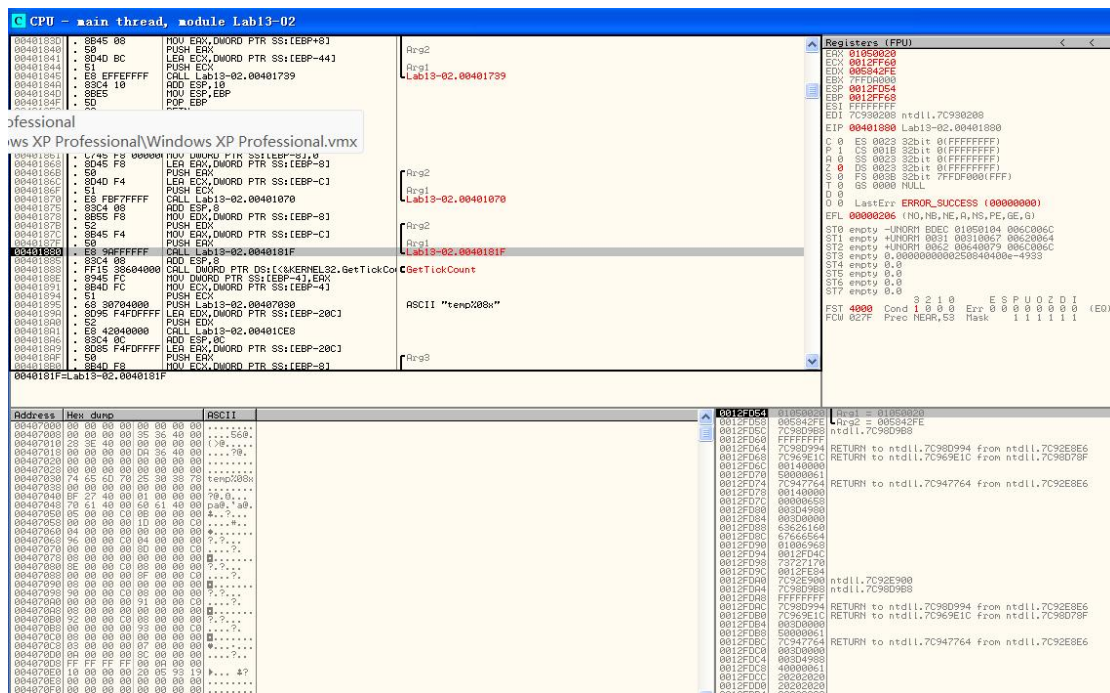
### 问题 7

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？

答：能

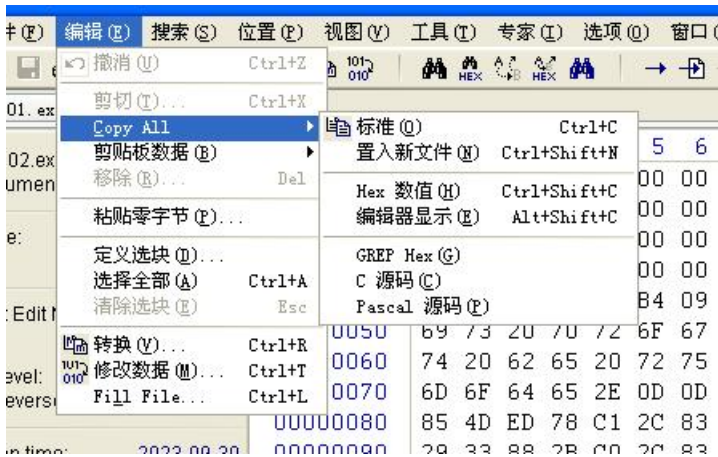
分析过程：

使用 OllyDbg 加载该程序，在加密之前先设置一个断点，并且保证在这时缓冲区 Buffer 中已经保存了要加密的屏幕截图信息，因此可以选择 0x00401880 作为断点，运行，命中断点后，此时堆栈上的参数表示的就是加密缓冲区 Buffer 和长度，查看堆栈中的值，如下：

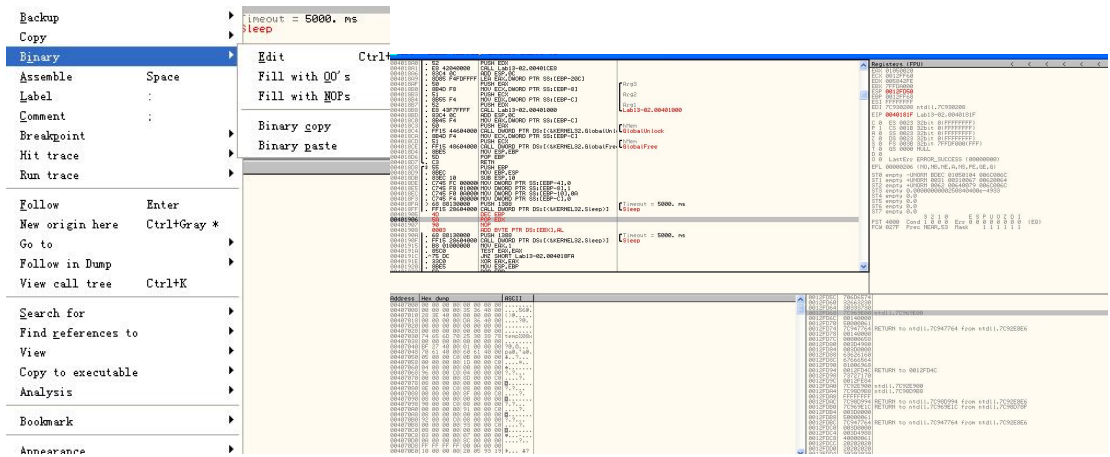


然后运行 WinHex，打开一个恶意代码创建的加密文件，选择 Edit→CopyAll→Hex Values，并在 OllyDbg 中将复制的内容 粘贴到缓冲区中。设置第二个断点为 0x0040190A 处，即在第一个文件写入后，继续运行程序，命中断点，在当前目录中找与先前 创建文件具有相同命名约定的新文件，修改扩展名为.bmp，就恢复出了屏幕截图。（在这里我们假设加密和解密用的是同样的函数）

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 中可以看出 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）。



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

### LAB13-3

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？

答：在执行动态分析时看到一些随机内容，推测其被加密了。而在输出字符串中找不到什么可能被加密的内容。

分析过程：

首先使用 strings 工具简单查看一下有哪些字符串



LocalFree	GetACP	Object not Initialized
WriteConsoleA	GetOEMCP	Data not multiple of Block Size
GetStdHandle	GetProcAddress	Empty key
lstrlenA	LoadLibraryA	Incorrect key length
FormatMessageA	SetStdHandle	Incorrect block length
GetLastError	LCMapStringA	.?AUexception@E
WriteFile	LCMapStringW	.?AUios_base@std@E
ReadFile	GetStringTypeA	.?AU?\$basic_ios@DU?\$char_traits@E@std@E@std@E
WaitForSingleObject	GetStringTypeW	.?AU?\$basic_istream@DU?\$char_traits@E@std@E@std@E
CreateThread	6:E	.?AU?\$basic_ostream@DU?\$char_traits@E@std@E@std@E
CreateProcessA	c:E	.?AU?\$basic_streambuf@DU?\$char_traits@E@std@E@std@E
CloseHandle	G:E	.?AU?\$basic_filebuf@DU?\$char_traits@E@std@E@std@E
DuplicateHandle	dJE	.?AU?\$basic_ios@GU?\$char_traits@E@std@E@std@E
GetCurrentProcess	'KE	.?AU?\$basic_istream@GU?\$char_traits@E@std@E@std@E
CreatePipe	TK@	.?AU?\$basic_ostream@GU?\$char_traits@E@std@E@std@E
KERNEL32.dll	8LE	.?AU?\$basic_filebuf@GU?\$char_traits@E@std@E@std@E
wsprintfA	y\E	.?AU?\$basic_istream@GU?\$char_traits@E@std@E@std@E
USER32.dll	z\@	.?AU?\$basic_ostream@GU?\$char_traits@E@std@E@std@E
WSASocketA	9d@	.?AU?\$basic_filebuf@GU?\$char_traits@E@std@E@std@E
WS2_32.dll	@KE	.?AU?\$basic_streambuf@GU?\$char_traits@E@std@E@std@E
MultiByteToWideChar	CDEFGHIJKLMNOPQRSTUVWXYZABcdefghij	.?AUruntime_error@std@E
RaiseException	ERROR: API = %s.	.?AUfailure@ios_base@std@E
GetCommandLineA	error code = %d.	.?AUfacet@locale@std@E
GetVersion	message = %s.	.?AU_Locimp@locale@std@E
RtlUnwind	ReadFile	.?AUlogic_error@std@E
HeapFree	WriteConsole	.?AUlength_error@std@E
TerminateProcess	ReadConsole	.?AUout_of_range@std@E
HeapReAlloc	WriteFile	.?AUtype_info@E
HeapAlloc	dir	JE
HeapSize	DuplicateHandle	'@
SetUnhandledExceptionFilter	DuplicateHandle	
UnhandledExceptionFilter	DuplicateHandle	
GetModuleFileNameA	CloseHandle	
FreeEnvironmentStringsA	CloseHandle	
FreeEnvironmentStringsW	GetStdHandle	
WideCharToMultiByte	cmd.exe	
GetEnvironmentStrings	CloseHandle	
GetEnvironmentStringsW	CloseHandle	
SetHandleCount	CreateThread	
GetStartupInfoA	CreateThread	
HeapDestroy	ijklmnopqrstuvwxyz	
HeapCreate	www.practicalmalwareanalysis.com	
VirtualFree	L"a	
SetFilePointer	d"a	
FlushFileBuffers	Object not Initialized	
VirtualAlloc	Data not multiple of Block Size	
IsBadWritePtr	Empty key	
IsBadReadPtr	Incorrect key length	
IsBadCodePtr		
GetCPInfo		

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  
 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 1.7462400	192.168.159.1	192.168.159.2	60S	92 Standard query 0x20B1 A www.practicalmalwareanalysis.com
5 1.0000700	192.168.159.1	192.168.159.255	UDP	305 Source port: 54915 Destination port: 54915
6 1.74692000	192.168.159.131	192.168.159.2	60S	92 Standard query 0x20B1 A www.practicalmalwareanalysis.com
7 1.78898000	192.168.159.2	192.168.159.131	60S	138 Standard query response 0x20B1 0x0000 practicalmalwareanalysis.com A 192.0.78.25 A 192.0.78.24
8 1.78901400	192.168.159.2	192.168.159.131	60S	138 Standard query response 0x20B1 0x0000 practicalmalwareanalysis.com A 192.0.78.24 A 192.0.78.25
9 1.78905500	192.168.159.131	192.0.78.25	TCP	62 nfinag > marguna-http [SRV] Seq=0 Wt=64240 Len=0 MSH=1460 SACK_PERM=1
10 1.90630200	192.168.159.1	192.168.159.255	UDP	305 Source port: 54915 Destination port: 54915
11 1.90638500	192.168.159.1	192.168.159.255	UDP	305 Source port: 54915 Destination port: 54915
12 1.90644400	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_22:12:13:a (00:0c:29:22:12:13:a), Dst: VMware_FcId2:b6 (00:15:56: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)	
Main Name System (Query)	

3	00	50	56	fc	d1	b6	00	0c	59	22	21	3a	08	00	41	00	.Pr.....?.....E.
3	00	4e	05	1e	00	00	80	15	51	a8	c0	a8	9f	83	c0	a8	.M.....U.....
3	9f	03	04	01	00	35	00	3a	c3	50	10	70	01	00	00	01	.....S.....P.....
3	00	00	00	00	00	03	27	77	27	28	70	72	61	61	74		.....www.pract
3	69	63	61	6c	6d	61	6c	77	61	72	61	6e	61	6c	79		icalmalwareanalysis.com.....
3	73	69	73	03	63	6f	6d	00	00	01	00	01					



可以看见出现了刚刚分析得到的 url

进行基础动态分析技术, 可以看到恶意代码尝试解析域名 [www.practicalmalwareanalysis.com](http://www.practicalmalwareanalysis.com),

并且向外 连接那个主机的 TCP 端口 8910。如果使用 Netcat 向这个连接发送内容, 可以看

到恶意代码用一些随机内容作为 响应, 但是看起来都是不可读的乱码 (可能被加密了),

如果我们从 Netcat 端终止这个连接, 可以看到:

Go

ERROR: API = ReadConsole.

error Code = 0.

message = The operation completed successfully.

与输出字符串进行对比, 看到了域名和看起来像前面提到的出错信息的字符串。

## 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?

答: 可能是异或

分析过程:

搜索字符串 xor, 共得到 191 条结果, 忽略用于清空寄存器的和库函数中的操作, 发现了 6

个包含 xor 指令的函数, 为了便于后续识别, 我们将其重命名。而进入这些函数查看 xor 指

令的上下文, 并不能识别出是哪一种类型的加密。

.text:00401135	sub_401082	xor	eax, eax	: jumptable
.text:0040123C	sub_401082	xor	eax, eax	
.text:00401310	sub_4012E9	xor	ecx, ecx	
.text:00401341	sub_40132B	xor	eax, eax	
.text:00401357	sub_40132B	xor	eax, eax	
.text:0040136D	sub_40132B	xor	eax, eax	
.text:004014A5	StartAddress	xor	eax, eax	
.text:004014BB	StartAddress	xor	eax, eax	
.text:00401873	sub_4015B7	xor	eax, eax	
.text:004019A5	_main	xor	eax, eax	
.text:00401A53	sub_401A50	xor	eax, eax	
.text:00401D51	sub_401AC2	xor	edx, edx	
.text:00401D69	sub_401AC2	xor	eax, eax	
.text:00401D88	sub_401AC2	xor	eax, eax	
.text:00401DA7	sub_401AC2	xor	eax, eax	
.text:00401EDB	sub_401AC2	xor	eax, edx	
.text:00401ED8	sub_401AC2	xor	eax, edx	
.text:00401EF3	sub_401AC2	xor	eax, edx	
.text:00401F08	sub_401AC2	xor	eax, edx	
.text:00401F13	sub_401AC2	xor	edx, eax	
.text:00401F58	sub_401AC2	xor	eax, [esi+edx*4+414h]	
.text:00401FB1	sub_401AC2	xor	edx, [esi+ecx*4+414h]	
.text:00402028	sub_401AC2	xor	edx, ecx	
.text:00402046	sub_401AC2	xor	edx, ecx	
.text:00402064	sub_401AC2	xor	edx, ecx	
.text:00402070	sub_401AC2	xor	ecx, edx	

有非常多的地方都使用了 xor，一共是有 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 熵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？

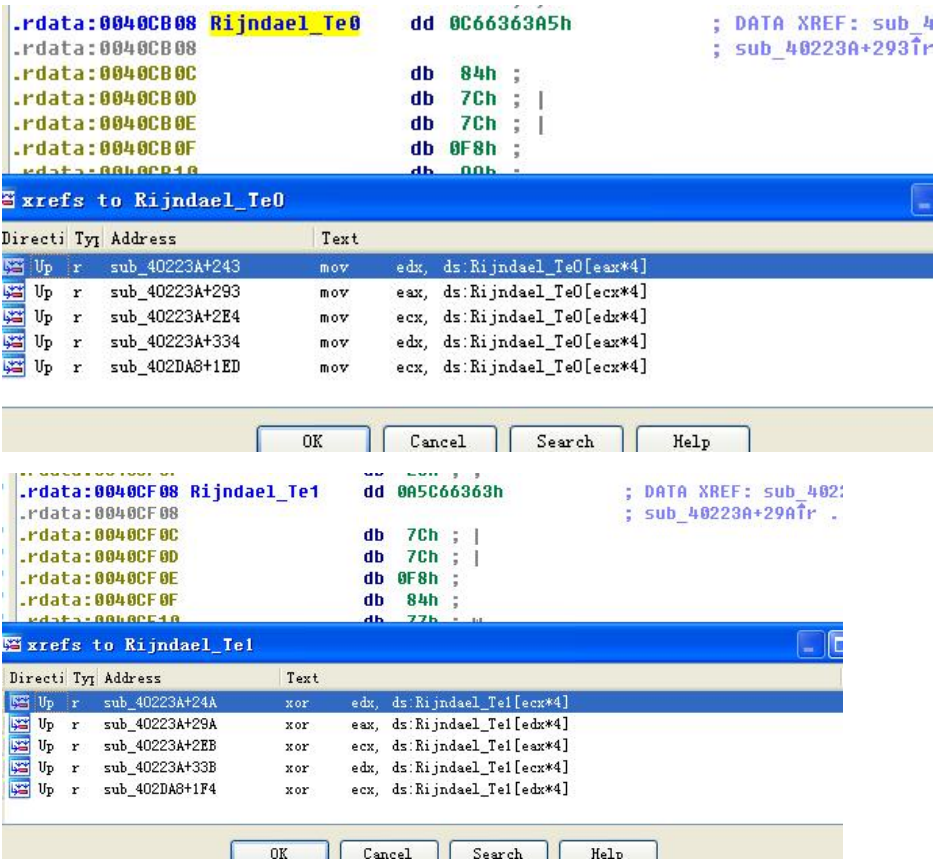
答：使用这三个插件都识别出了 AES 算法，它与搜索 xor 识别出的 6 个函数相关。而 IDA 的熵插件还识别出了一个自定义的 Base64 索引字符串。具体见分析过程

分析过程：

使用 IDA 的 FindCrypt2 插件进行查找

```
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



```
.rdata:0040D308 Rijndael_Te2 dd 63A5C663h ; DATA XREF: sub_40223A+21r
.rdata:0040D308 ; sub_40223A+2AD↑r ...
```

xrefs to Rijndael\_Te2

Directi	Ty	Address	Text
Up	r	sub_40223A+25C	xor edx, ds:Rijndael_Te2[edx*4]
Up	r	sub_40223A+2AD	xor eax, ds:Rijndael_Te2[ecx*4]
Up	r	sub_40223A+2FE	xor ecx, ds:Rijndael_Te2[edx*4]
Up	r	sub_40223A+34D	xor edx, ds:Rijndael_Te2[edx*4]
Up	r	sub_402DA8+218	xor ecx, ds:Rijndael_Te2[edx*4]

```
.rdata:0040D708 Rijndael_Te3 dd 6363A5C6h ; DATA XREF: sub_40223A+26C↑r
.rdata:0040D708 ; sub_40223A+2BD↑r ...
```

xrefs to Rijndael\_Te3

Directi	Ty	Address	Text
Up	r	sub_40223A+26C	xor edx, ds:Rijndael_Te3[ecx*4]
Up	r	sub_40223A+2BD	xor eax, ds:Rijndael_Te3[edx*4]
Up	r	sub_40223A+30D	xor ecx, ds:Rijndael_Te3[ecx*4]
Up	r	sub_40223A+35D	xor edx, ds:Rijndael_Te3[ecx*4]
Up	r	sub_402DA8+239	xor ecx, ds:Rijndael_Te3[edx*4]

```
.rdata:0040DB08 Rijndael_Td0 dd 51F4A750h ; DATA XREF: sub_4027ED↑r
.rdata:0040DB08 ; sub_4027ED+248↑r ...
```

xrefs to Rijndael\_Td0

Directi	Ty	Address	Text
Up	r	sub_4027ED+248	mov edx, ds:Rijndael_Td0[edx*4]
Up	r	sub_4027ED+298	mov eax, ds:Rijndael_Td0[ecx*4]
Up	r	sub_4027ED+2E9	mov ecx, ds:Rijndael_Td0[edx*4]
Up	r	sub_4027ED+339	mov edx, ds:Rijndael_Td0[edx*4]
Up	r	sub_403166+1F0	mov ecx, ds:Rijndael_Td0[ecx*4]

```
.rdata:0040DF08 Rijndael_Td1 dd 5051F4A7h ; DATA XREF: sub_4027ED↑r
.rdata:0040DF08 ; sub_4027ED+24F↑r ...
```

xrefs to Rijndael\_Td1

Directi	Ty	Address	Text
Up	r	sub_4027ED+24F	xor edx, ds:Rijndael_Td1[ecx*4]
Up	r	sub_4027ED+29F	xor eax, ds:Rijndael_Td1[edx*4]
Up	r	sub_4027ED+2F0	xor ecx, ds:Rijndael_Td1[ecx*4]
Up	r	sub_4027ED+340	xor edx, ds:Rijndael_Td1[ecx*4]
Up	r	sub_403166+1F7	xor ecx, ds:Rijndael_Td1[edx*4]

```
.rdata:0040E307 db 42h ; B
.rdata:0040E308 Rijndael_Td2 dd 0A75051F4h ; DATA XREF: sub_4027ED+261↑r
.rdata:0040E308 ; sub_4027ED+2B2↑r ...
```

xrefs to Rijndael\_Td2

Directi	Ty	Address	Text
Up	r	sub_4027ED+261	xor edx, ds:Rijndael_Td2[edx*4]
Up	r	sub_4027ED+2B2	xor eax, ds:Rijndael_Td2[ecx*4]
Up	r	sub_4027ED+303	xor ecx, ds:Rijndael_Td2[edx*4]
Up	r	sub_4027ED+352	xor edx, ds:Rijndael_Td2[edx*4]
Up	r	sub_403166+21B	xor ecx, ds:Rijndael_Td2[edx*4]

```
.rdata:0040E708 Rijndael_Td3 dd 0F4A75051h ; DATA XREF: sub_4027ED+2C2↑r
.rdata:0040E708 ; sub_4027ED+312↑r ...
```

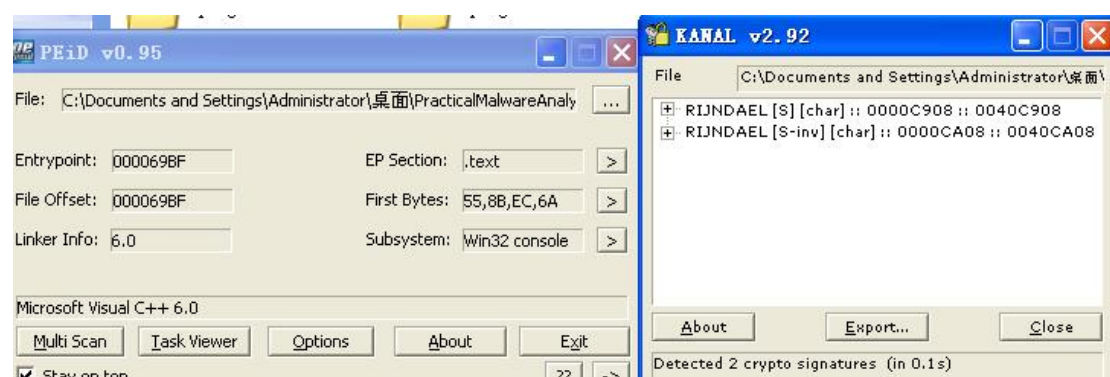
xrefs to Rijndael\_Td3

Directi	Ty	Address	Text
Up	r	sub_4027ED+271	xor edx, ds:Rijndael_Td3[ecx*4]
Up	r	sub_4027ED+2C2	xor eax, ds:Rijndael_Td3[edx*4]
Up	r	sub_4027ED+312	xor ecx, ds:Rijndael_Td3[ecx*4]
Up	r	sub_4027ED+362	xor edx, ds:Rijndael_Td3[ecx*4]
Up	r	sub_403166+23C	xor ecx, ds:Rijndael_Td3[edx*4]

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

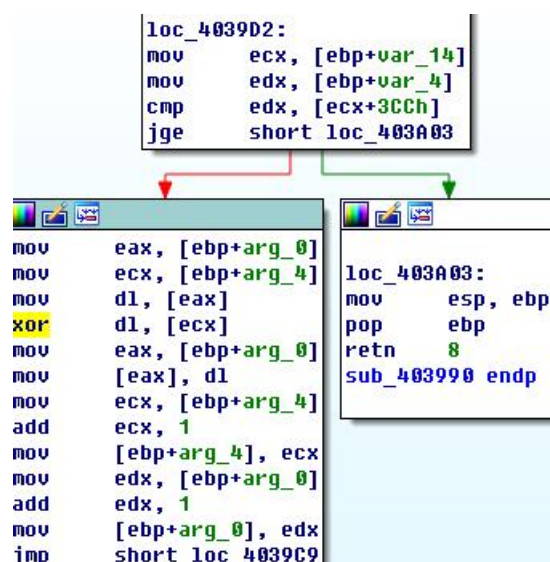


使用 PEiD 工具的插件可以发现



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

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

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

```
loc_401B59:
mov     eax, [ebp+var_60]
mov     ecx, [ebp+arg_8]
mov     [eax+3C8h], ecx
mov     edx, [ebp+var_60]
mov     eax, [ebp+arg_C]
mov     [edx+3CCh], eax
mov     ecx, [ebp+var_60]
mov     edx, [ecx+3CCh]
push    ecx                ; void *
push    edx                ; size_t
call    _memcpy
add     esp, 0Ch
mov     eax, [ebp+arg_4]
push    eax                ; void *
mov     ecx, [ebp+var_60]
add     ecx, 3D4h
push    ecx                ; void *
call    _memcpy
add     esp, 0Ch
mov     edx, [ebp+var_60]
mov     eax, [edx+3CCh]
push    eax                ; size_t
push    ecx                ; void *
mov     ecx, [ebp+arg_4]
push    ecx                ; void *
mov     edx, [ebp+var_60]
add     edx, 3F4h
push    edx                ; void *
call    _memcpy
add     esp, 0Ch
mov     eax, [ebp+var_60]
mov     ecx, [eax+3C8h]
mov     [ebp+var_64], ecx
cmp     [ebp+var_64], 10h
jz      short loc_401BCB
```

查看 x1 的交叉应用可以发现

```
ext:00401879      push    ebp
ext:0040187A      mov     ebp, esp
ext:0040187C      sub     esp, 1ACh
ext:00401882      push    10h                ; int
ext:00401884      push    10h                ; int
ext:00401886      push    offset unk_413374 ; void *
ext:0040188B      push    offset aijklmnopqrstuv ; "ijklmnopqrstuvwx"
ext:00401890      mov     ecx, offset unk_412EF8
ext:00401895      call    sub_401AC2
ext:0040189A      lea     eax, [ebp+WSAData]
ext:004018A0      push    eax                ; lpWSAData
ext:004018A1      push    202h                ; wVersionRequested
ext:004018A6      call    ds:WSAStartup
ext:004018AC      mov     [ebp+var_194], eax
ext:004018B2      cmp     [ebp+var_194], 0
ext:004018B9      jz      short loc_4018C5
ext:004018BB      mov     eax, 1
ext:004018C0      jmp     loc_4019A7
```

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

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？

答： AES 加密算法和自定义的 Base64 加密算法。



```

.text:00401414      mov     eax, [ebp+nNumberOfBytesToWrite]
.text:0040141A      push    eax
.text:0040141B      lea     ecx, [ebp+var_FE8]
.text:00401421      push    ecx
.text:00401422      lea     edx, [ebp+Buffer]
.text:00401428      push    edx
.text:00401429      mov     ecx, offset unk_412EF8
.text:0040142E      call    sub_40352D
.text:00401433      push    0 ; lpOverlapped
.text:00401435      lea     eax, [ebp+NumberOfBytesWritten]
.text:0040143B      push    eax ; lpNumberOfBytesWritten
.text:0040143C      mov     ecx, [ebp+nNumberOfBytesToWrite]
.text:00401442      push    ecx ; nNumberOfBytesToWrite
.text:00401443      lea     edx, [ebp+var_FE8]
.text:00401449      push    edx ; lpBuffer
.text:0040144A      mov     eax, [ebp+var_BE0]
.text:00401450      mov     ecx, [eax+4]
.text:00401453      push    ecx ; hFile
.text:00401454      call    ds:WriteFile
.text:0040145A      test    eax, eax
.text:0040145C      jnz     short loc_40146B
.text:0040145E      push    offset aWriteconsole ; "WriteConsole"
.text:00401463      call    sub_401256

```

分析过程:

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

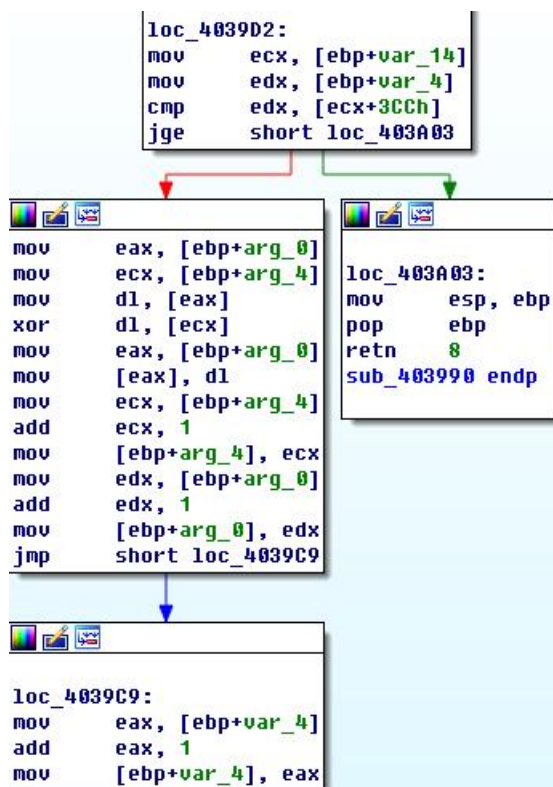
问题 5

对于每一种加密技术，它们的密钥是什么？

答：AES 的密钥是 ijklmnopqrstuvwxyz

自定义的 Base64 加密的密钥是

CDEFGHIJKLMNOPQRSTUVWXYZABcdefghijklmnopqrstuvwxyzab0123456789+/-



分析过程:

首先查看 AES 算法，我们已经知道，函数 s\_xor2 和 s\_xor4 与 AES 加密有关，s\_xor3 和 s\_xor5 与 AES 解密函数有关。然后查看剩下的两个函数 s\_xor1 和 s\_xor6，寻找其中的关联。

首先查看 s\_xor6 的代码，可以看到其 xor 操作在一个循环中，变量 arg\_0 指向被加密的原缓冲区，arg\_4 指向用来提供异或值的缓冲区。查看该函数的交叉引用，可以看到 s\_xor6 与 AES 加密函数 s\_xor2 和 s\_xor4 相关。

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

```
push    offset off_412240
lea     ecx, [ebp+var_10]
call    ??0exception@@QAE@ABQ8DQZ ; exception::exception(char const * const &)
push    offset unk_410858
lea     edx, [ebp+var_10]
push    edx
call    _CxxThrowException@8 ; _CxxThrowException(x,x)
```

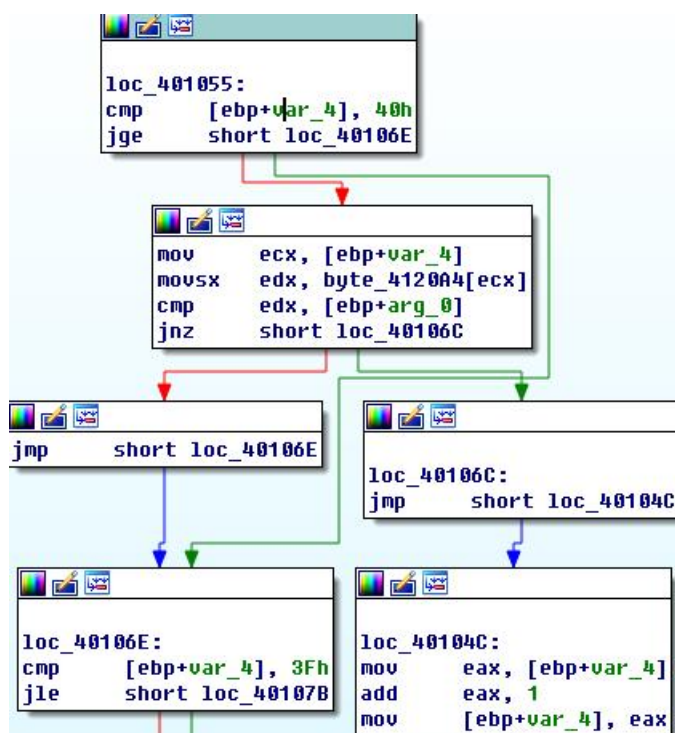
在 x1 中有一个地方会提示空密钥，而这个提示是基于 arg0 的

```
push    ebp
mov     ebp, esp
sub     esp, 68h
push    esi
mov     [ebp+var_60], ecx
cmp     [ebp+arg_0], 0
jnz     short loc_401AF3
```

回到调用他的地方查看参数

```
push    ebp
mov     ebp, esp
sub     esp, 1ACh
push    10h ; int
push    10h ; int
push    offset unk_413374 ; void *
push    offset aIjklmnopqrstuv ; "ijklmnopqrstuvwx"
mov     ecx, offset unk_412EF8
call    sub_401AC2
lea     eax, [ebp+WSAData]
push    eax ; lpWSAData
push    202h ; wVersionRequested
call    ds:WSAStartup
mov     [ebp+var_194], eax
cmp     [ebp+var_194], 0
jz      short loc_4018C5
```

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



事实上我们查看 s\_xor1 的代码，发现其非常复杂，有很多的 cmp 操作引出的分支，当传给该函数的参数不正确时，开始走向不同的分支，而不同的分支有不同的错误信息，包括：空密钥、不正确的密钥长度、不正确的块长度，由此推测该函数用来进行密钥的初始化。

首先比较了 arg\_0 与 0，如果该参数为空，对应的错误信息是 Empty key，因此 arg\_0 就是密钥。到 s\_xor1 被调用的地方，可以看到该参数是 ijklmnoparstuvwx，这就是被用来 AES 加密的字符串。查看

该函数的交叉引用，看到在 s\_xor1 被调用之前，存在一个对 unk\_412EF8 的引用，将这个偏移量作为参数 传入 s\_xor1 函数。查看 unk\_412EF8 的其他引用，发现 0x401429 是该偏移量载入 ECX 的地方之一，而它在调用 函数 sub\_403745 之前被载入到 ECX。由此 unk\_412EF8 是一个表示 AES 加密器的 C++ 对象，并且 s\_xor1 是加密器的初始化函数。

而 base64 的密钥就是 strings 中看见的那个字符串，也就是 loc\_401055

再查看 Base64 加密算法。检查对字符串

CDEFGHIJKLMNOPQRSTUVWXYZABcdefghijklmnopqrstuvwxyzab0123456789+ / 的引用，看到这个字符串在 0x0040103F 中被使用，该函数用索引 var\_4 查询这个字符串并且调用函数 sub\_401082，将解密的字符串分成 4 字节的块，可以看到调用 sub\_401082 的上下文，发现在其前后分别调用了 ReadFile 和 WriteFile，推测函数 sub\_401082 就是该恶意代码自定义的 Base64 解码函数。

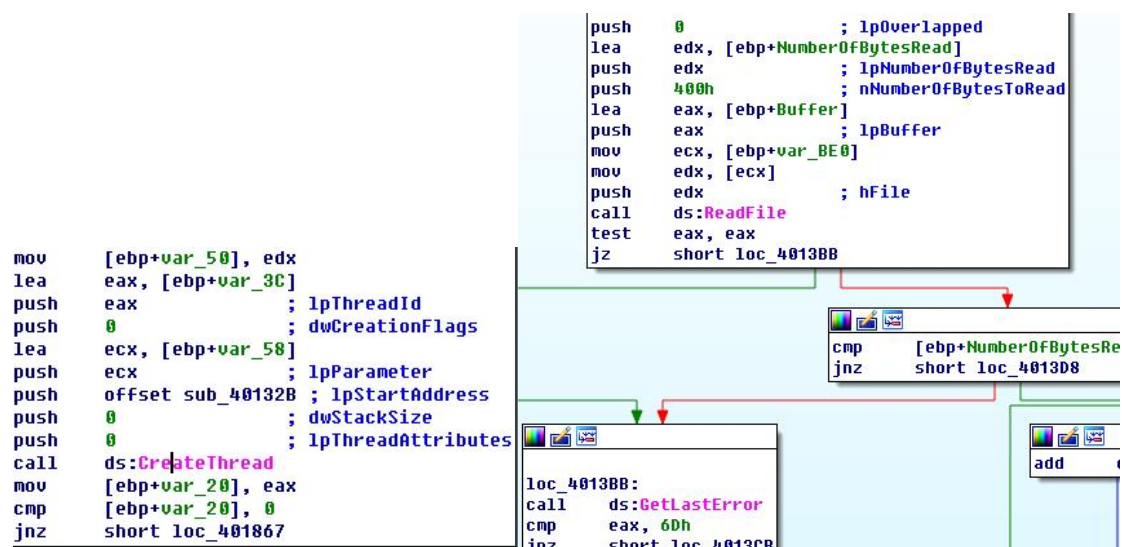
## 问题 6

对于加密算法，它的密钥足够可靠吗？另外你必须知道什么？

答： 对于 AES 算法，解密还需要密钥之外的变量，包括密钥生成算法、密钥大小、操作模式，以及一些常量的初始化等；而对于这个自定义的 Base64 加密，当前已知的索引字符串已经足够了。

## 问题 7 恶意代码做了什么？

答： 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。该恶意代码用 AES 用于在写入网络套接字前加密 shell 命令的输出结果，而这个自定义的 Base64 用来加密传入的命令。



分析过程：

在 main 函数中我们可以看见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

首先这里有一个 `readfile` 的函数，这个函数的参数是 `var_BE0`，而这个 `BE0` 来自于 `arg_0`，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，

往上走可以发现这里其实就是这个函数的一个参数。

```
loc_401914:
mov     ecx, [ebp+var_198]
mov     edx, [ecx+0Ch]
mov     eax, [edx]
mov     ecx, [eax]
mov     dword ptr [ebp+name.sa_data+2], ecx
push    22CEh                ; hostshort
call    ds:htons
mov     word ptr [ebp+name.sa_data], ax
mov     [ebp+name.sa_family], 2
push    10h                  ; namelen
lea     edx, [ebp+name]
push    edx                  ; name
mov     eax, [ebp+s]
push    eax                  ; s
call    ds:connect
mov     [ebp+var_194], eax
cmp     [ebp+var_194], 0FFFFFFFh
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

我们已经知道加密函数在 sub\_40132b 中被调用，而这个函数又被 sub\_4015b7 引用，跟进，

一个新线程，sub\_40132b 作为参数，

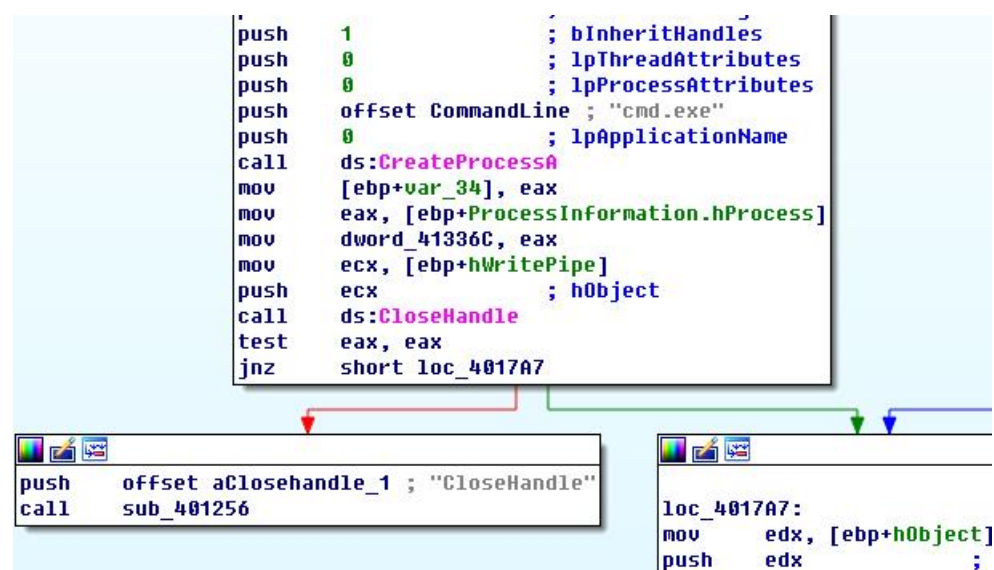
个新线程的 参数保存在 lpParameter,

些参数的具体意义。该函数首先调用



是传入该函数的唯一参数，我们已经知道该参数是 var\_58。之后又调用了 WriteFile，其参数 hFile 为 var\_BE0+4，也就是 var\_54，我们看到 var\_58 和 var\_18 持有一个管道的句柄，并且这个管道的与一个 shell 命令的输出相连接，令 hSourceHandle 通过函数 DuplicateHandle 复制到 shell 命令的标准输出和标准错误，这条 shell 命令由通过调用 CreateProcess 来启动。

回溯这个频繁被用到的参数 var\_54，可以看到它是 sub\_4015b7 的唯一参数，在 main 函数中我们可以知道这个 参数是[ebp+s]，它是调用函数 connect 后创建的一个网络套接字。



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 CreatePorcessA 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，这两个应该是一个先后顺序：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 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？

答：

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

python 脚本如下：

```

import string
import base64

result = ""

ciphter_content = "CDEFGHIJKLMNOPQRSTUVWXYZABabcdefghijklmnopqrstuvwxyz0123456789+/" standard_b64 =
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"

```

```

ciphter_text = "BlnaEi=="
for each_ch in ciphter_text:
    if each_ch in ciphter_content:
        result += standard_64[string.find(ciphter_content, str(each_ch))]
    elif each_ch == '=':
        s += '='
result = base64.decodestring(result)
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 JS...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./.

```

之后我们将在 wireshark 当中抓包的结果利用 AES 进行分析，可以得到解密结果。

```

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

```

## 四、实验结论及心得体会

本次实验，虽然我们最终没有 yara 规则和 ida python 代码的编写，但是我们在运用分析的过程当中，大量考虑到了字符串，其中有相当一部分字符串是加密解密函数的明显标志。这些都会是 yara 规则和 ida python 脚本编写的重要工具，其中我们在 lab13-03.exe 文件当中尝试的对相应的文件进行的调用，以及 AES 解码的过程，更是类似 ida python 工具编写的有效方法。

在本次实验当中，我对恶意代码工具的运用有了更加深刻的理解，对使用恶意代码分析工具解决实际的问题，有了很明确的认知。本次课程与学习过的《密码学》课程相互联系起来，可以有效的将密码学的知识以及加解密程序分析得到很好的认知。希望我能在后续对这两门课程的内容学习以及应用上能够有更好的发展，在计算机病毒加解密工作上能有更加充分的认识，期待去的更大的进步。