恶意代码实验十三报告

学号：　　姓名：　　专业：

一. 实验环境

1. 已关闭病毒防护的 Windows10
2. VMware 16PRO + WindowsXP

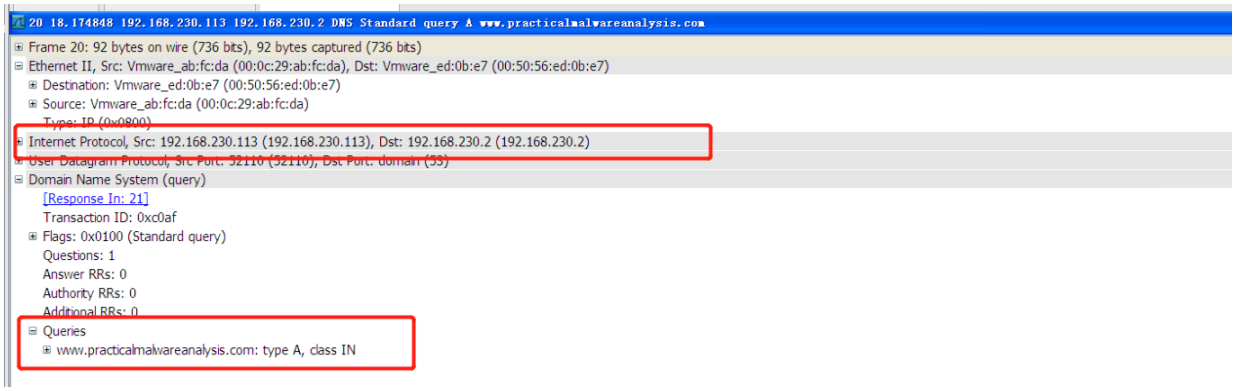
二、 实验工具

IDAPro、Dependency Walker、Wireshark、Ollydbg、Strings、ProcessMonitor、Process Explorer、RegShot

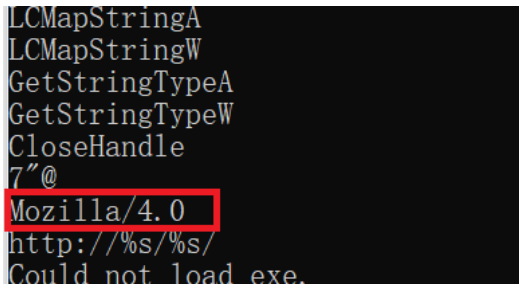
三. Lab13-01

具体分析

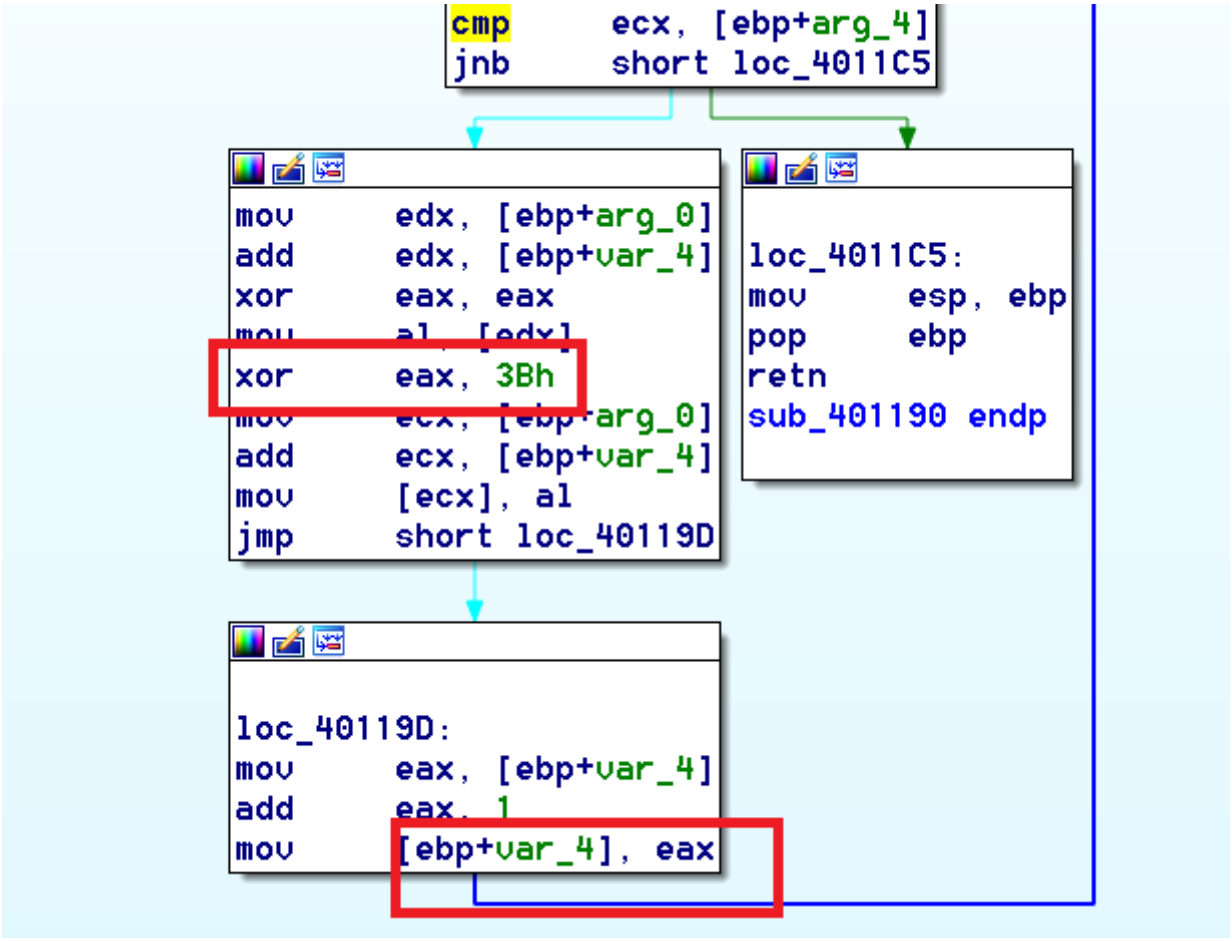
1. 首先我们可以运行Lab13-01.exe来分析其具体的行为，并通过wireshark进行抓包分析。我们可以通过结果简单发现他请求了一个网站www.practicalmalwareanalysis.com:



2. strings工具检查Lab13-01.exe字符串，可以发现我们并不能观察到其访问的网址以及那个奇怪的乱码字符串，所以我们可以猜测有一些重要字符串被加密了



3. 进入IDAPro，通过Alt+t快捷键全局搜索XOR指令，可以发现有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，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：

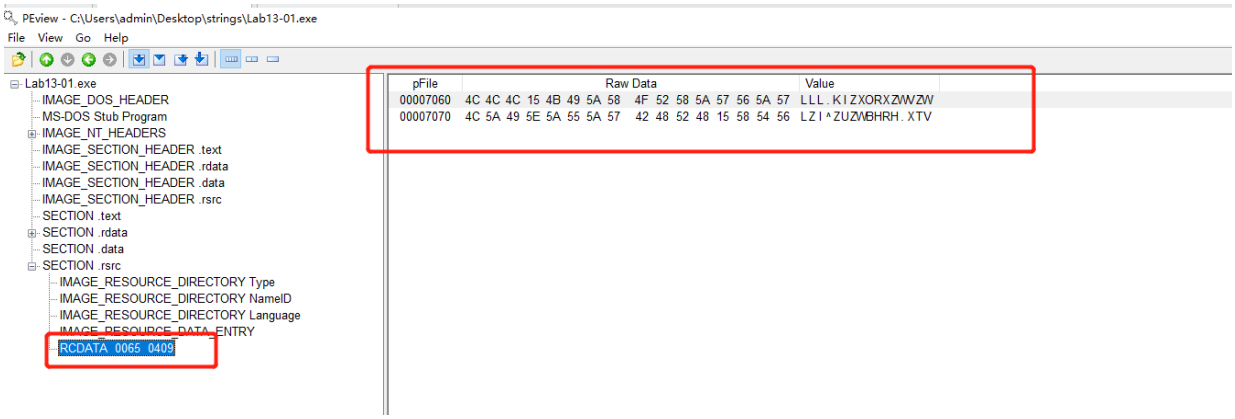


这里就是一个循环进行异或的操作，每次递增的是var_4中的内容（在这里就是长度），异或的内容是 arg_0，那么这里很明显就是一个异或加密的操作了，密钥是3Bh，所以这个sub_401190函数就是进行 异或加密的函数

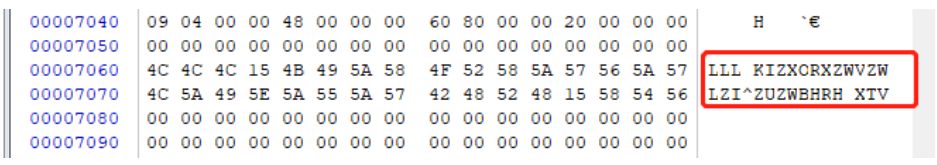
4. 然后我们需要知道究竟是谁调用了这个函数，并且参数是什么字符串：函数sub401300是调用上面函数的唯一函数。跟踪它调用解密函数之前的代码块，我们依次看到调用GetModuleHandleA、FindResourceA、SizeofResource、GlobalAlloc、LoadResource以及LockResource。调用解密函数之前，恶意代码会对资源做一些处理。在这些资源相关的函数中，我们应该研究让我们找到资源数据的函数FindResourceA。

```
.text:00401339 loc_401339:                             ; CODE XREF: sub_401300+23↑j
.text:00401339                 push    0Ah             ; lpType
.text:0040133B                 push    65h             ; lpName
.text:0040133D                 mov     eax, [ebp+hModule]
.text:00401340                 push    eax             ; hModule
.text:00401341                 call    ds:FindResourceA
.text:00401347                 mov     [ebp+hResInfo], eax
.text:0040134A                 cmp     [ebp+hResInfo], 0
.text:0040134E                 jnz     short loc_401357
.text:00401350                 xor     eax, eax
.text:00401352                 jmp     loc_4013E9
.text:00401357 ; ---------------------------------------------------------------------------
.text:00401357
.text:00401357 loc_401357:                             ; CODE XREF: sub_401300+4E↑j
.text:00401357                 mov     ecx, [ebp+hResInfo]
.text:0040135A                 push    ecx             ; hResInfo
.text:0040135B                 mov     edx, [ebp+hModule]
.text:0040135E                 push    edx             ; hModule
.text:0040135F                 call    ds:SizeofResource
.text:00401365                 mov     [ebp+dwBytes], eax
.text:00401368                 mov     eax, [ebp+dwBytes]
.text:0040136B                 push    eax             ; dwBytes
.text:0040136C                 push    40h             ; uFlags
.text:0040136E                 call    ds:GlobalAlloc
.text:00401374                 mov     [ebp+var_4], eax
.text:00401377                 mov     ecx, [ebp+hResInfo]
.text:0040137A                 push    ecx             ; hResInfo
.text:0040137B                 mov     edx, [ebp+hModule]
.text:0040137E                 push    edx             ; hModule
.text:0040137F                 call    ds:LoadResource
.text:00401385                 mov     [ebp+hResData], eax
.text:00401388                 cmp     [ebp+hResData], 0
.text:0040138C                 jnz     short loc_401392
.text:0040138E                 xor     eax, eax
.text:00401390                 jmp     short loc_4013E9
.text:00401392 ; ---------------------------------------------------------------------------
.text:00401392
```

5. 再上图关于FindResourceA函数：其中lpType是0xA，它表示资源数据是应用程序预定义的还是原始数据。lpName既可以是一个名字，也可以是一个索引号。本例中，它是一个索引号。由于函数引用了ID为101的资源，于是通过PEview工具去寻找该资源，也就是索引号为0x65的资源，查找也比较简单，打开资源节找到非常醒目的RCDATA就可以找到目标字符串，在资源节偏移0x7060的位置



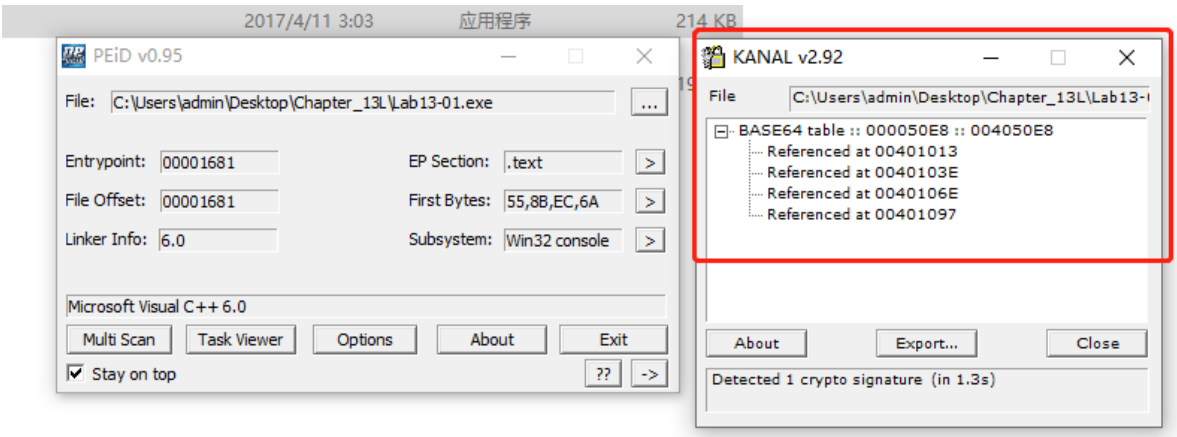
6. 然后我们使用WinHEX对这部分资源进行转换：



7. 选择转转方式为XOR然后输入要异或的数字3B即可，可以观察到下图中以加密形式存储的字符串www.praticalmalwareanalysis.com：

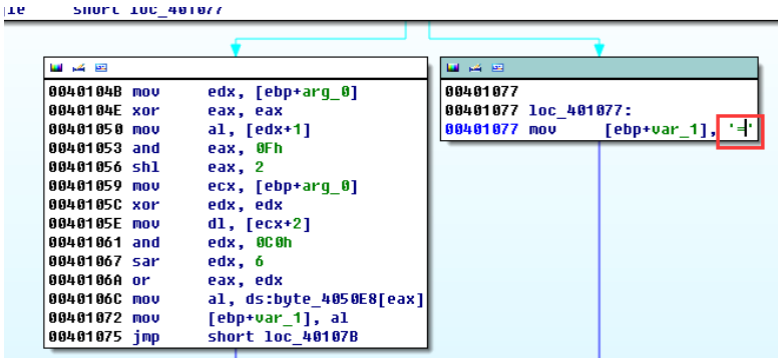
```
00007030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 01 00
00007040  09 04 00 00 48 00 00 00  60 80 00 00 20 00 00 00      H    `€
00007050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00007060  77 77 77 2E 70 72 61 63  74 69 63 61 6C 6D 61 6C  www.practicalmal
00007070  77 61 72 65 61 6E 61 6C  79 73 69 73 2E 63 6F 6D  wareanalysis.com
00007080  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00007090  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
000070A0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
```

8. 其中有两个字符串也可能被加密，我们发现了域名，但是没有发现GET请求字符串。为了找到GET字符串，我使用PEiD的KANAL插件，它在0x004050F8处找到一个Base64编码表。



9. 我们找到在基本静态分析时找到的典型的Base64编码表的位置，发现他有四个交叉引用，并且全部位于sub401000函数

   我们可以看到，图中右侧的框中有一个字符=的交叉引用。这证实了我们的判断:sub401000与Base64编码相关，因为在Base64加密中以“=’作为填充字符。调用本函数的函数(4010B1) 是真正的Base64加密函数。它的目的是将源字符串划分成3个字节的块，并且将3个字节块传递给本函数，从而将传入的3个字节加密成4个字一些线索使得这更加明显:函数开头使用strlen查找源字符串的长度，在外部循环(代码块loc 401100)的开头与3比较 ，当本函数返回结果后，在内部写循环的开头与4比较。因此我们得出结论: 前面分析的解密函数就是Base64编码的主函数，他用源字符和Base64加密转换的目的缓冲区作为参数



10. 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

```
00401000 arg_8= dword ptr  10h
00401000
00401000 push    ebp
00401001 mov     ebp, esp
00401003 push    ecx
00401004 mov     eax, [ebp+arg_0]
00401007 xor     ecx, ecx
00401009 mov     cl, [eax]
0040100B sar     ecx, 2
0040100E mov     edx, [ebp+arg_4]
00401011 mov     al, ds:byte_4050E8[ecx]
00401017 mov     [edx], al
00401019 mov     ecx, [ebp+arg_0]
0040101C xor     edx, edx
0040101E mov     dl, [ecx]
00401020 and     edx, 3
00401023 shl     edx, 4
00401026 mov     eax, [ebp+arg_0]
00401029 xor     ecx, ecx
0040102B mov     cl, [eax+1]
0040102E and     ecx, 0F0h
```

11. 使用F5键进入到C语言代码的形式，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。

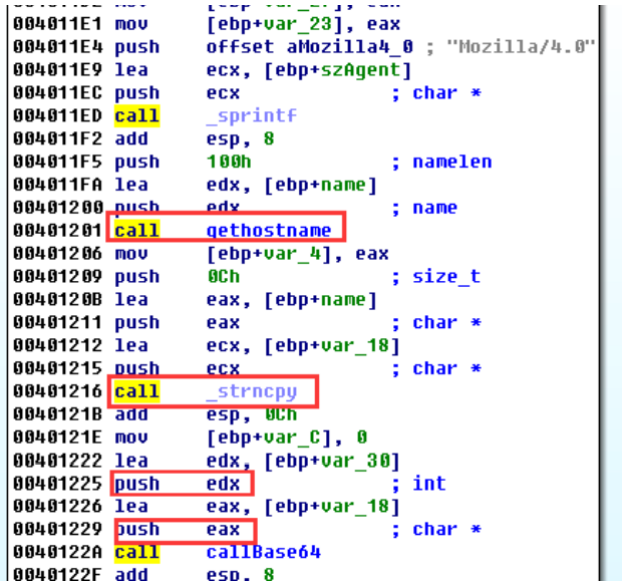
```
signed int j; // [sp+8h] [bp-14h]@10
char v7[4]; // [sp+Ch] [bp-10h]@5
char v8[4]; // [sp+10h] [bp-Ch]@10
size_t v9; // [sp+14h] [bp-8h]@1
size_t v10; // [sp+18h] [bp-4h]@1

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;
    }
```

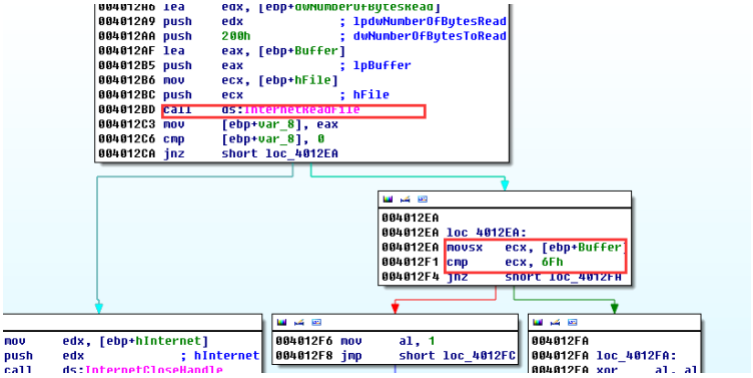
12. 查看循环体内部，可以看见循环体内部就是每次取出来三个字符，然后利用base64进行解密的操作

```
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];
  }
}
```

13. 回到刚刚的函数体，查看一下交叉引用，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。



14. 观察beacon中的剩余代码，我们看到它使用了WinINet(InternetOpenA 、InternetOpenUrlA和InternerReadFile)，返回数据的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



15. 概括来说，这个恶意代码发送通信信号beacon，让攻击者知道它在正常运行。恶意代码用加密的（有可能截断）主机名作为标识符发送一个特定的通信信号，当接收到一个特定的回应后，则终止。

习题解答

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？**

   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， 另外一个是GET请求路径，它看起来像aG9zdG5hbWUtZm9v。

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?**

   地址004011B8处的xor指令是sub_401190函数中的一个单字节XOR加密循环的指令。

3. **恶意代码使用什么密钥加密，加密了什么内容?**

   单字节XOR加密使用字节0x3B。用101索引原始的数据源解密的XOR加密缓冲区的内容是www.practicalmalwareanalysis.com。

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?**

   用插件PEiD KANAL和IDA熵，可识别出恶意代码使用标准的Base64编码字符串:
   ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/

5. **什么类型的加密被恶意代码用来发送部分网络流量?**

   标准的Base64编码用来创建GET请求字符串。

6. **Base64编码函数在反汇编的何处?**

   Base64加密函数从0x004010B1处开始。

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?**

   Base64加密前，Lab13-01.exe复制最大12个字节的主机名，这使得GET请求的字符串的最大字符个数是16.

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==)?**

   如果主机名小于12个字节并且不能被3整除，则可能使用填充字符。

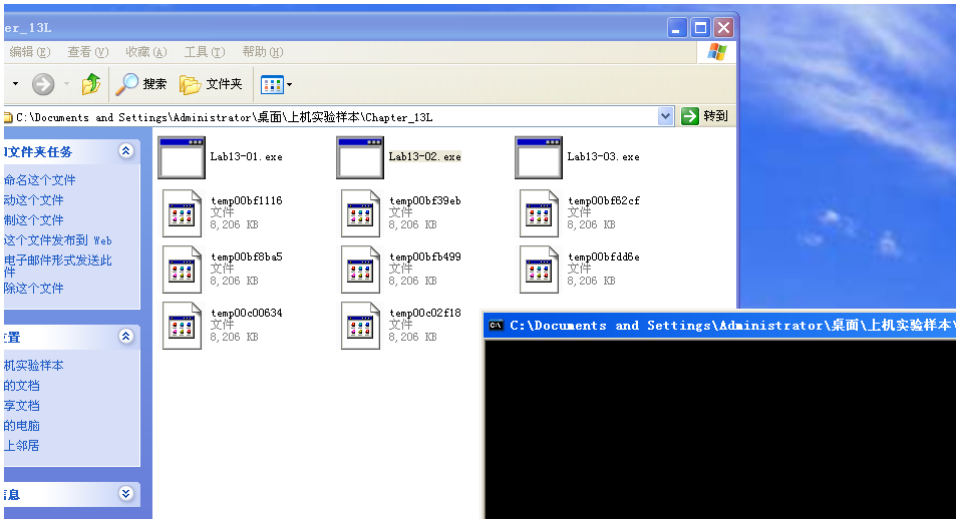
9. **这个恶意代码做了什么?**

   Lab13-01.exe用加密的主机名发送一个特定信号，直到接收特定的回应后退出

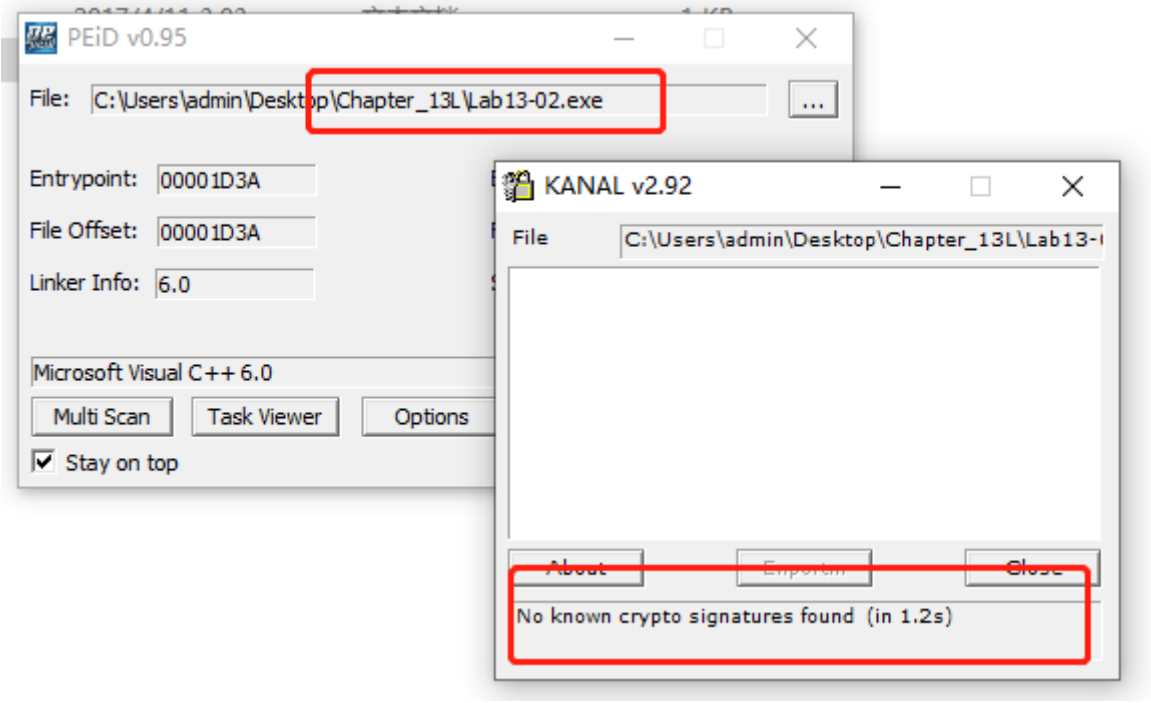
四. Lab13-02

具体分析

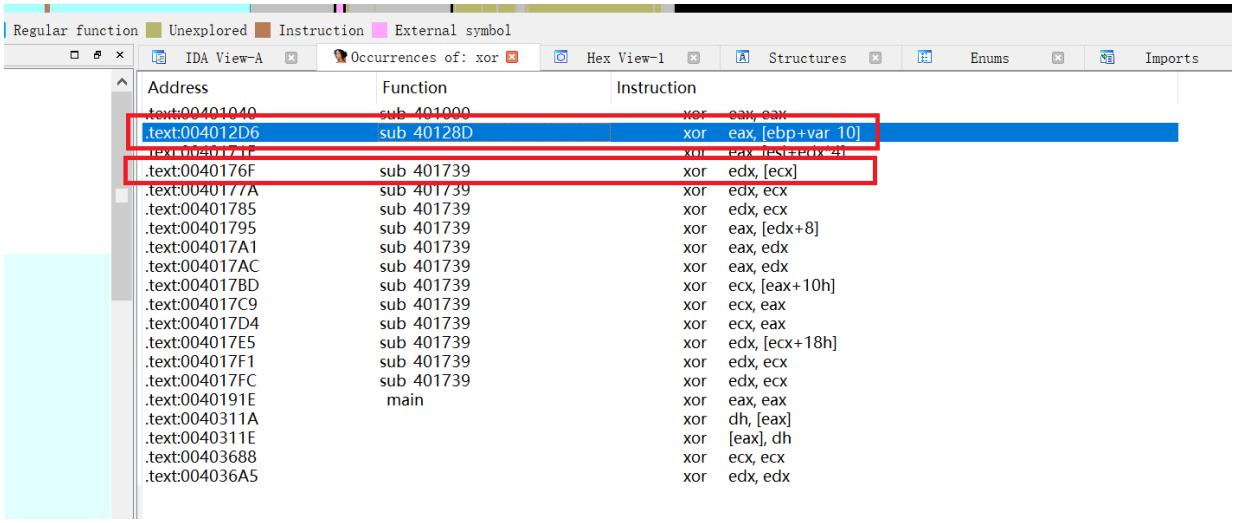
1. 启动恶意代码，我们可以观察到他会在原始目录下以固定的时间间隔创建一些新的文件，这些文件相当大（几兆大小）并且文件似乎包含一些随机的数据，文件名以temp开始并且以一些随机的字符结束



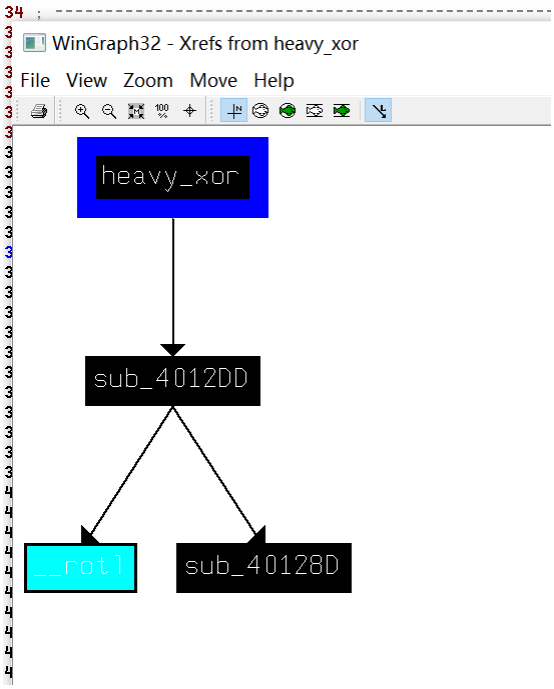
2. 使用peid扫描该可执行文件未发现可以被轻易检测的加密证据：



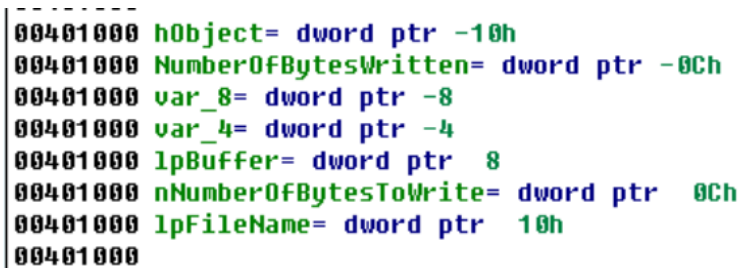
3. 进入IDAPro，通过Alt+t快捷键全局搜索XOR指令，去除掉一些库函数的引用、寄存器清零等无特殊含义的异或指令调用，我们可以找到两个很有价值的异或指令：分别位于004012D6和0040176F，尤其要注意位于0x0040171F的xor指令，我们看到它位于一个函数中，但是由于很少使用，这个函数并没有被自动识别。在0x00401570定义函数导致创建包含了前面孤立的xor指令函数。这个未使用的函数也与同组内可能的加密函数有关联。



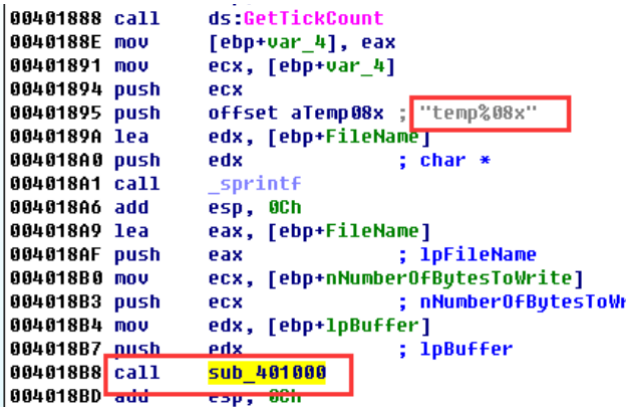
4. 因为sub401739拥有非常多的xor指令，我们将sub 491739称为Heavy_xor。而sub40128D拥有一条xor指令，我们将sub40128D称为SINGLE_xor。heavyxor带有4个参数，它是一个单循环，除了xor指令以外，拥有包含多个包含SHL和SHR指令的代码块。看一下heavy_xor调用的函数，我们发现singlexor与heavy_xor相关，因为singlexor的调用者也被heavy_xor调用。



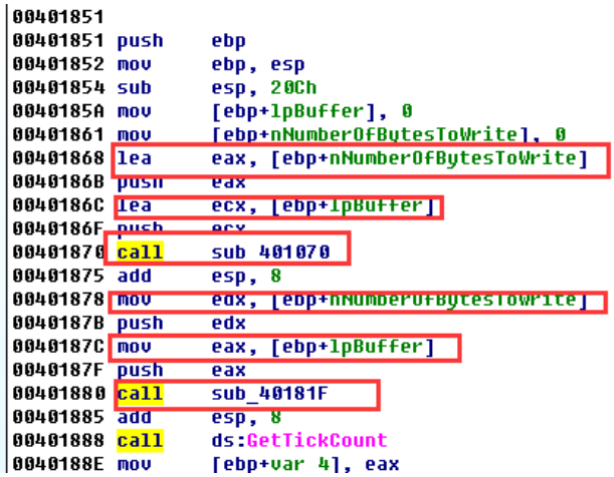
5. 根据刚刚对WriteFile函数的分析我们可以知道主要是sub_401000函数调用了这个函数，查看一下这个函数，可以看见这个函数的参数有如上几个，并且通过参数的名称就可以看出有一个参数是要写入的byte数 量，一个是文件名，还有一个是缓冲区的指针。



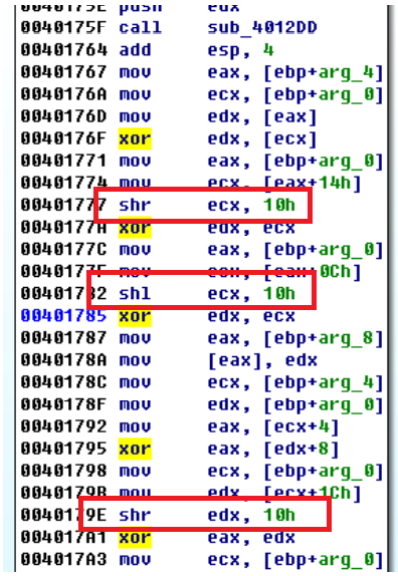
6. 找到调用这个函数的位置，看见出现了刚刚创建的文件的文件名的一部分，同时我们注意到，在上面有一个函数是 GetTickCount ， 也就是获取系统启动了的时间，那么猜测这里的文件名就是创建这样文件名的一个文件。



7. 往上查看，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。

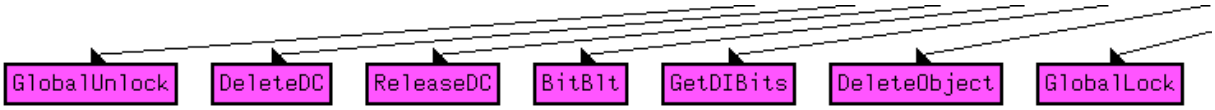


8. 进入到第二个函数发现他就是刚刚调用xor函数的函数，并且密钥为10h，那么这个加密函数就在反汇编中的sub_401739



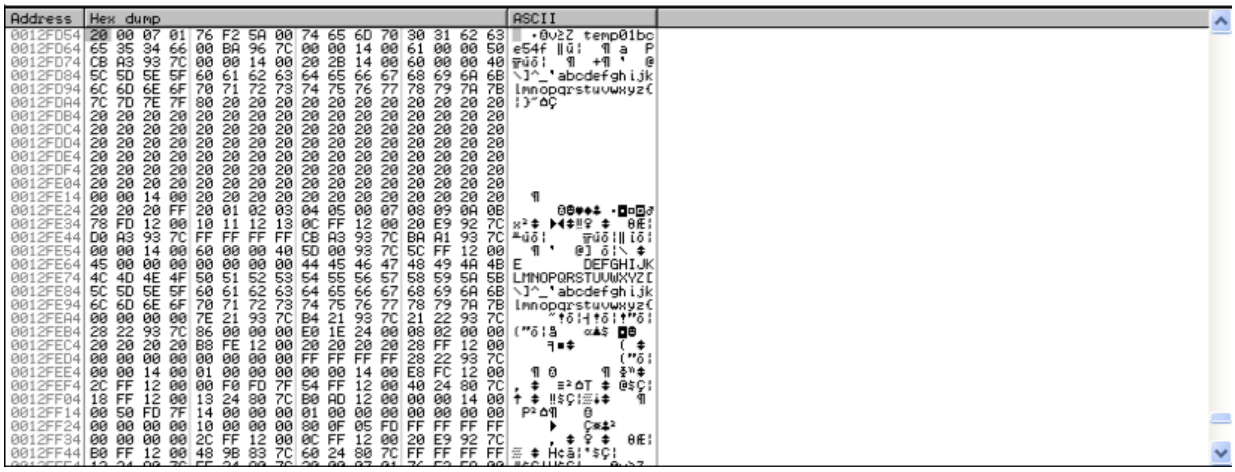
9. 根据刚刚的分析，我们认为这个加密算法使用的就是一个简单的异或加密，对于异或加密操作来说，解密和加密是使用的同一套流程，所以解密的时候同样是使用10h进行异或操作即可。 或者是在恶意代码获取了缓冲区的内容后，将缓冲区里的内容进行修改，改成以前得到的加密文件，然 后让他再次执行xor的操作，就能够达到解密的效果。

10. 为了确定heavyxor的确是一个加密函数，让我们来看一下它与写入到磁盘上temp文件有何种关系我们找到了数据写入到磁盘的位置，接下来，我们确定加密并且写入磁盘的原始内容。如前面提到的一样，函数getContent(在0x00401070)似乎获取一些内容。看一下getContent，我们看到一个拥有多个系统函数的代码块。我们可以根据他的系统函数调用列表进行猜测，一种较好的猜测是:这个函数试图抓屏。值得注意的是，GetDesktopwindow获取覆盖整个屏幕桌面窗口的一个句柄，函数BitBlt和GetDIBits(粗体所示)获取位图信息并将它们复制到缓冲区。因此，我们得出结论，恶意代码反复抓取用户的桌面并且将加密版本的抓屏信息写入到一个文件



11. 比较正常的一种思路就是使用ollydbg进行解密，这时我们需要设置两个断点，第一个事在加密开始前，也就是在0x00401880作为断点，第二个断电我们可以设置在文件写入到内存后，也就是位于0x0040190A，此时我们可以根据观察寄存器和堆栈的值来进行解密。

12. 此时一个比较重要的值是esp，我们需要在真正的内存中取出对应位置的值，当我们找到对应位置时果然发现了一些数据，但这些数据我们是无法直接读取的，我们需要做的就是先把他们复制出来：



13. 然后我们可以点击十六机制的转存，随后点击ollydbg的运行键让程序运行到最后衹断点的位置。这个时候可以检查恶意代码目录中与之前产生的文件有相同名字的文件，并给他添加扩展名bmp再次打开他就可以观察到一张关于屏幕的截屏了

习题解答

1. **使用动态分析，确定恶意代码创建了什么？**

   启动恶意代码，我们可以观察到他会在原始目录下以固定的时间间隔创建一些新的文件，这些文件相当大（几兆大小）并且文件似乎包含一些随机的数据，文件名以temp开始并且以一些随机的字符结束。

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

   XOR搜索技术在sub_401570和sub_401739中识别了加密相关的函数，其他3中推荐的技术并没有发现什么。

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？**

   WriteFile调用之前可能会发现加密函数

4. **加密函数在反汇编的何处？**

   加密函数是sub_40181F

5. **从加密函数追溯原始的加密内容，原始加密内容是什么？**

   原内容是屏幕截图

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？**

   加密算法是不标准算法，并且不容易识别，最简单的方法是通过解密工具解密流量。

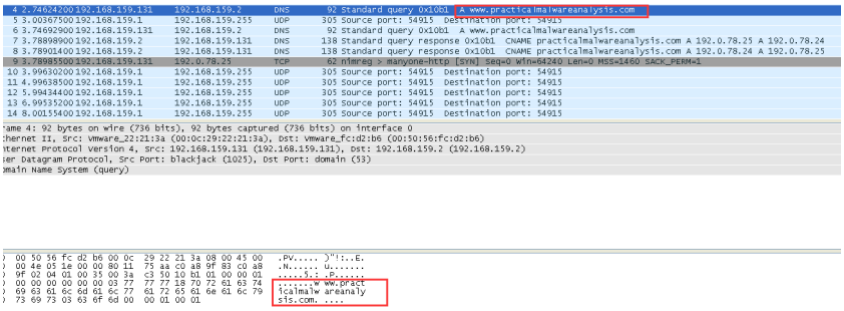
7. **使用解密工具，你是否能够恢复加密文件中的一个文件到原始文件？**

   见上述分析

五. Lab13-03

具体分析

1. 我们首先点击运行可执行文件并配置wireshark进行监听，我们发现应用程序对www.pracitcalmalwareanalysis进行了访问，也就是说，本恶意代码是具有网络特征的

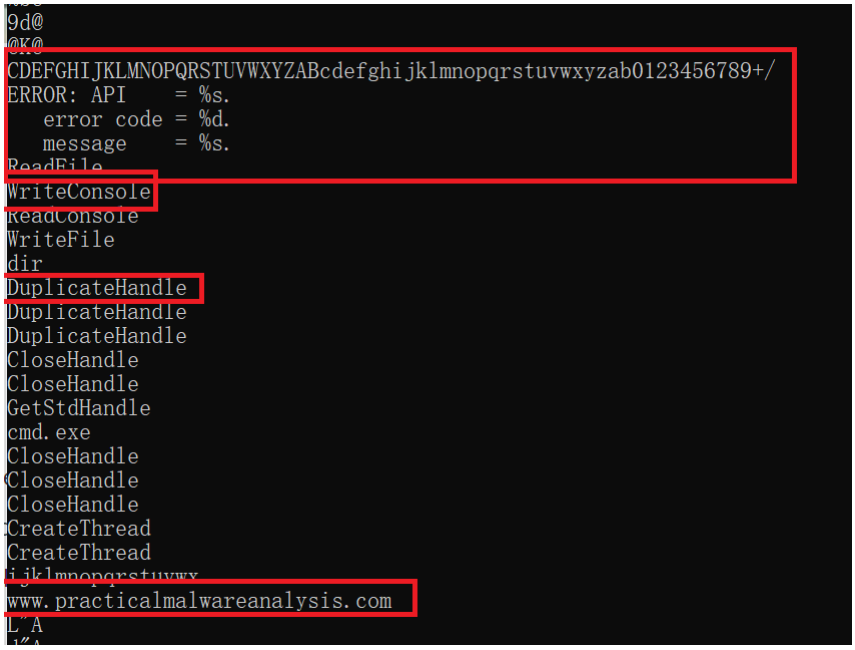


2. 通过String工具检查字符串，我们可以发现很对格式字符串以及和Base64相关的字符串，这说明有一些内容很有可能被加密了，但是网站的域名却没有被加密，同时我们观察这个很像Base64加密编码的字符序列，发现他并不是最常见的编码方式，因为AB和ab都被放在了最后，而他们却是以C和c开始的，并且，这里有一些比较新颖的win系统函数，我们可以通过查阅官方文档来了解这些函数的定义

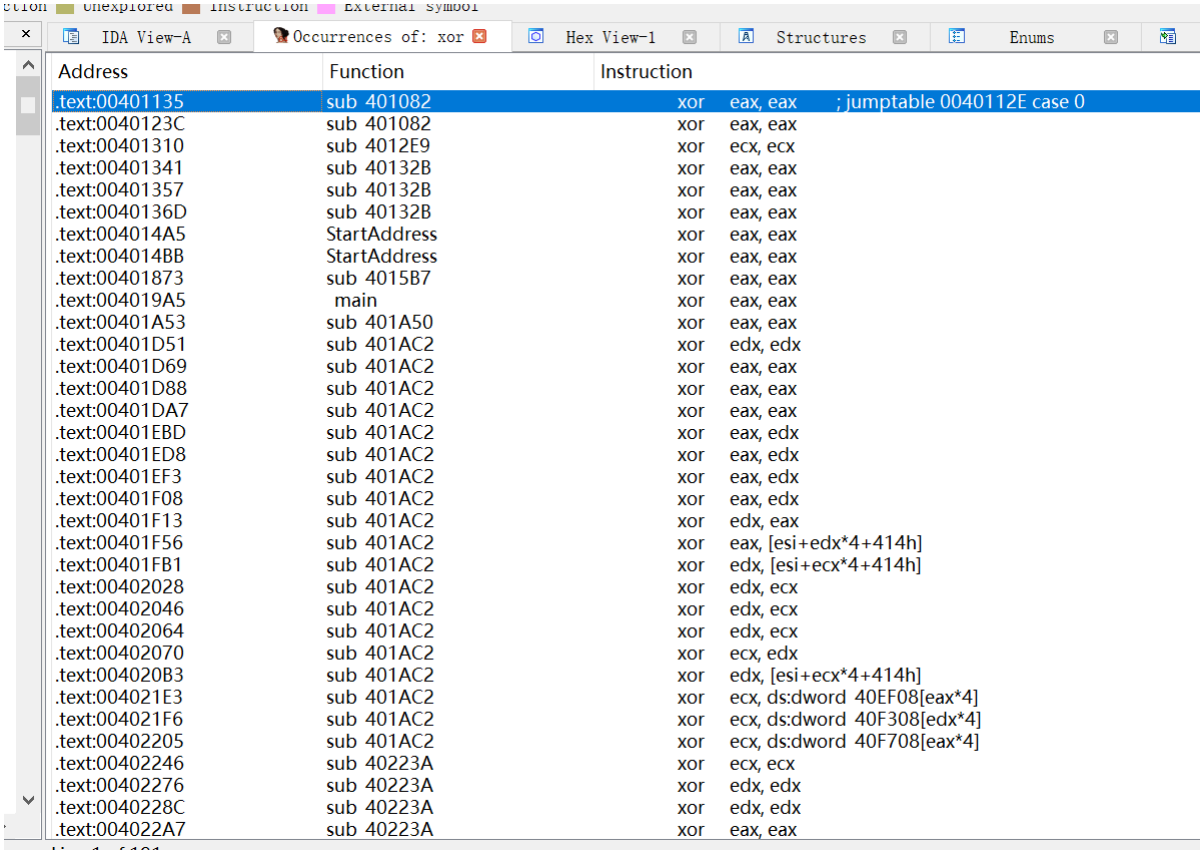
   WriteConsole：从当前光标位置开始，将字符串写入控制台屏幕缓冲区

   ReadConsole：从控制台输入缓冲区读取字符输入，并将其从缓冲区中删除。

   DuplicateHandle：复制对象句柄



3. 使用IDA查找xor指令以后可以发现有非常多的地方都使用了xor



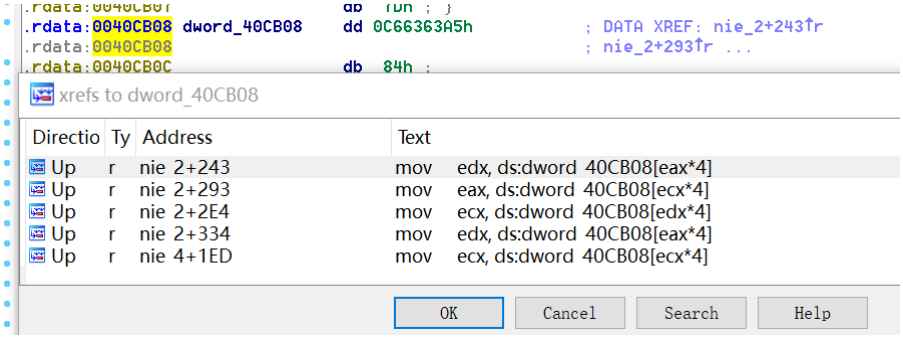
4. 检查一下xor指令并且去掉与寄存器清零和库函数相关的xor指令，我们发现6个包含xor指令的函数，那么这些函数可能存在有加密的行为，对这6个函数都进行重命名，分别为nie_1到nie_6。那么根据刚刚的分析，猜测有6处都有加密的可能，并且使用的加密方式可能是异或

分配的函数名	函数地址
s_xor1	00401AC2
s_xor2	0040223A
s_xor3	004027ED
s_xor4	00402DA8
s_xor5	00403166
s_xor6	00403990

5. 使用IDA的FindCrypt2插件进行查找

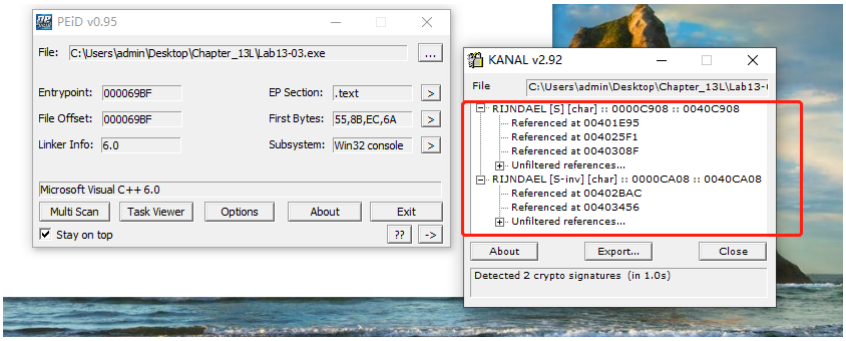
```
The initial autoanalysis has been finished.
40CB08: found const array Rijndael_Te0 (used in Rijndael)
40CF08: found const array Rijndael_Te1 (used in Rijndael)
40D308: found const array Rijndael_Te2 (used in Rijndael)
40D708: found const array Rijndael_Te3 (used in Rijndael)
40DB08: found const array 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.
```

6. 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



7. 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

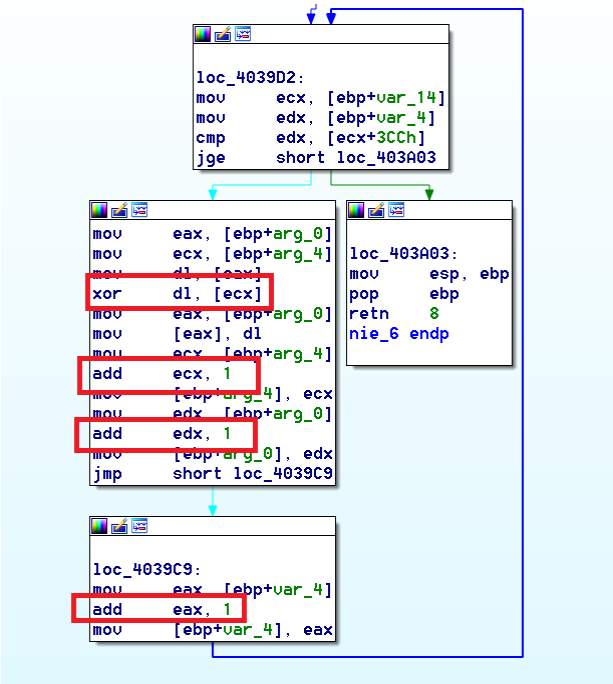
8. 使用peid的插件同样证实了使用AES算法



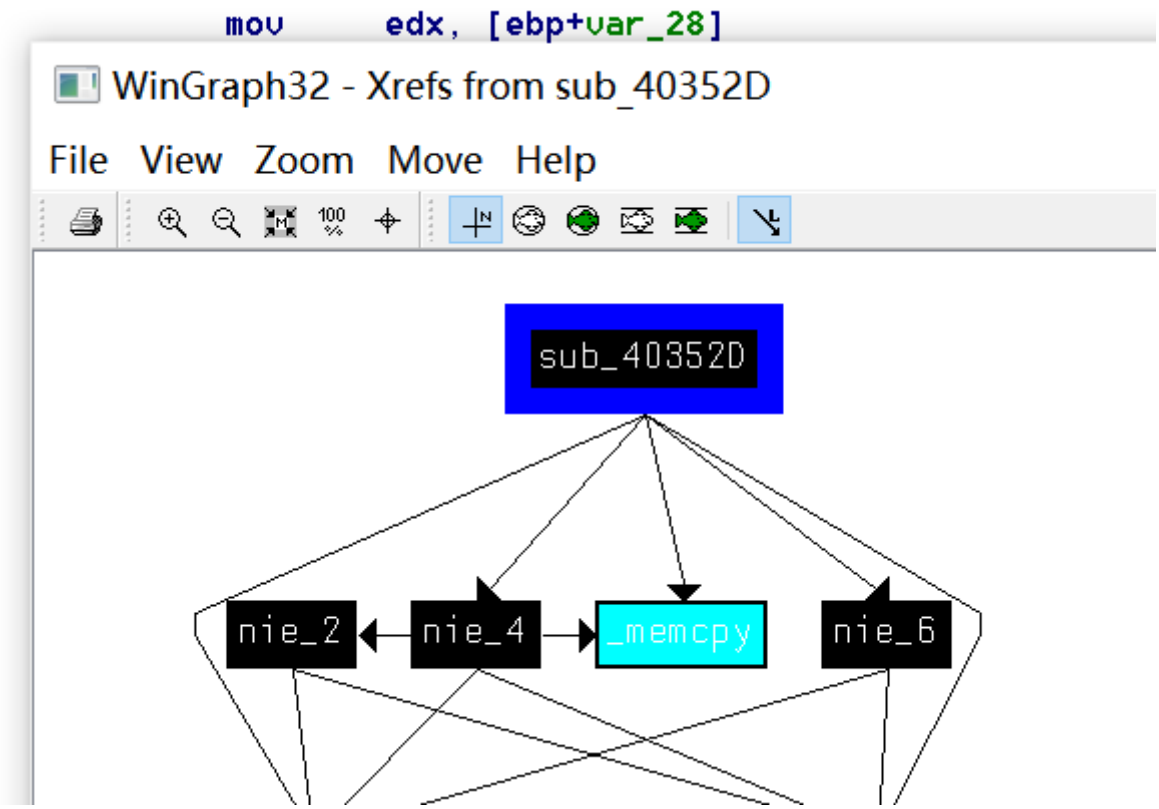
9. 然后我们使用IDA熵插件来显示熵比较高的位置，我们可以观察到在数据段0x0040C900开始的位置和AES中使用的S-box区域是相同的，也就是说通过不同的插件确认，我们可以确定该恶意代码使用了AES进行加密

```
.rdata:0040C909                 db  7Ch ; |
.rdata:0040C90A                 db  77h ; w
.rdata:0040C90B                 db  7Bh ; {
.rdata:0040C90C                 db 0F2h ;
.rdata:0040C90D                 db  6Bh ; k
.rdata:0040C90E                 db  6Fh ; o
.rdata:0040C90F                 db 0C5h ;
.rdata:0040C910                 db  30h ; 0
.rdata:0040C911                 db    1
.rdata:0040C912                 db  67h ; g
.rdata:0040C913                 db  2Bh ; +
.rdata:0040C914                 db 0FEh ;
.rdata:0040C915                 db 0D7h ;
.rdata:0040C916                 db 0ABh ;
.rdata:0040C917                 db  76h ; v
.rdata:0040C918                 db 0CAh ;
.rdata:0040C919                 db  82h ;
.rdata:0040C91A                 db 0C9h ;
.rdata:0040C91B                 db  7Dh ; }
.rdata:0040C91C                 db 0FAh ;
.rdata:0040C91D                 db  59h ; Y
.rdata:0040C91E                 db  47h ; G
.rdata:0040C91F                 db 0F0h ;
.rdata:0040C920                 db 0ADh ;
.rdata:0040C921                 db 0D4h ;
.rdata:0040C922                 db 0A2h ;
.rdata:0040C923                 db 0AFh ;
.rdata:0040C924                 db  9Ch ;
.rdata:0040C925                 db 0A4h ;
.rdata:0040C926                 db  72h ; r
.rdata:0040C927                 db 0C0h ;
.rdata:0040C928                 db 0B7h ;
.rdata:0040C929                 db 0FDh ;
.rdata:0040C92A                 db  93h ;
.rdata:0040C92B                 db  26h ; &
.rdata:0040C92C                 db  36h ; 6
.rdata:0040C92D                 db  3Fh ; ?
0000C909 0000000000040C909: .rdata:0040C909
```

10. 下图就是nie_6的xor指令相关的循环处理函数，其中第一个参数是一个指针，他指向了进行转换的原缓冲区，第二个参数也是一个指针，他指向了异或原数据的缓冲区，也就是该和谁异或。



11. 为了判断nie_6是否与其他加密函数之间有关联，我们检查它的交叉引用，可以发现调用nie_6的函数的起始地址为0x0040352D。下图展示了从0x0040352D开始，函数的交叉引用图：



12. 此时我们将关注点聚焦到nie_1上来，他并不直接服务于AES算法，但是他会处理好和加密相关信息的初始化并做出判断，包括如果密钥为空或者密钥的长度不正确都会被识别。而nie_1和其他函数的关系可以查看nie_1的调用函数，在nie_1被调用之前412EF8首先被调用，他将一个偏移量给予了nie_1，而412EF8在加密之前被载入到ECX，所以我们不难判断他其实是一个C++对象，或者说是一个AES的加密器，而nie_1接受了这个参数arg0，如果完成判断则空密钥的提升被发出，所以说arg0一定是一个密钥，而在main中nie_1的参数在0x401895被设置，这个字符串将被用于加密。

```
.text:00401AC2 arg_8           = dword ptr  10h
.text:00401AC2 arg_C           = dword ptr  14h
.text:00401AC2
.text:00401AC2                 push    ebp
.text:00401AC3                 mov     ebp, esp
.text:00401AC5                 sub     esp, 68h
.text:00401AC8                 push    esi
.text:00401AC9                 mov     [ebp+var_60], ecx
.text:00401ACC                 cmp     [ebp+arg_0], 0
.text:00401AD0                 jnz     short loc_401AF3
.text:00401AD2                 mov     [ebp+var_3C], offset aEmptyKey ; "Empty key"
.text:00401AD9                 lea     eax, [ebp+var_3C]
.text:00401ADC                 push    eax
.text:00401ADD                 lea     ecx, [ebp+var_38]
.text:00401AE0                 call    ??0exception@@QAE@ABQBD@Z ; exception::exception(char const * const &)
.text:00401AE5                 push    offset unk_410858
.text:00401AEA                 lea     ecx, [ebp+var_38]
.text:00401AED                 push    ecx
.text:00401AEE                 call    __CxxThrowException@8 ; _CxxThrowException(x,x)
.text:00401AF3 ; ---------------------------------------------------------------------------
```

13. 然后，接下来我们就需要明确这个AES代码在程序中做了一件什么事：0040132B的位置调用了加密函数，而这个加密函数发生在读文件前，在加密后完成了写函数。nie_1只在启动的时候被调用一次，他设置了密钥。并且Base64页参与了加密，检查对编码表的引用，这个字符串在0x0040103F函数中，函数索引编码表并且将解密后的字符串分成32bits的块，并且自定义了一个解码函数，同时也在读文件和写文件之间调用了它

```
.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
```

14. 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
```

15. 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
```

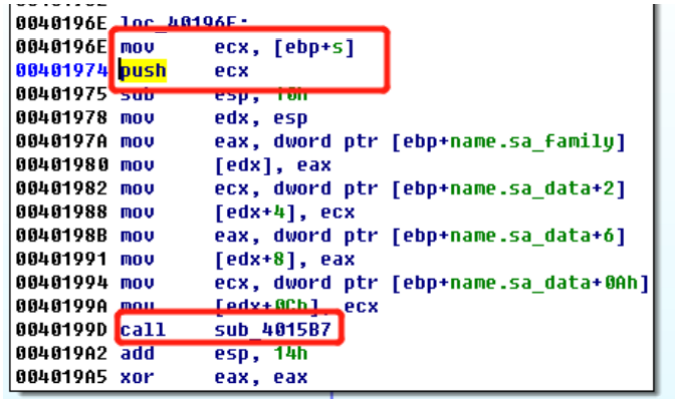
16. 线程内还有一个writefile的函数，这函数的参数

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

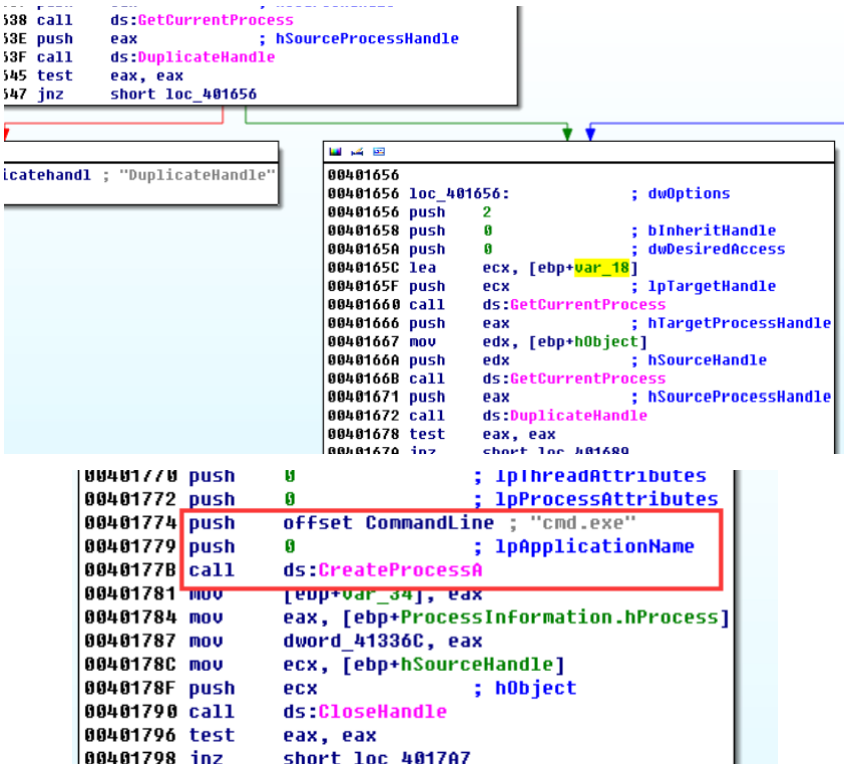
17. 从边上的注释可以看出来这个参数是arg_10

```
004015B7 ProcessInformation= _PROCESS_INFORM
004015B7 arg_10= dword ptr  18h
004015B7
004015B7 push    ebp
```

18. 往上走可以发现这里其实就是这个函数的一个参数



19. 回到函数内，我们可以发现如下图所示，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，这两个应该是有一个先后顺序：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



20. 解密算法：具体的解密算法参考书后给出的代码即可对其加密的内容进行解密，具体的Base64算法为：

```
import string
import base64

S=""
tab="CDEFGHIJKLMNOPORSTUVWXYZABcdefghiklmnopqrstuvwxyzab0123456789+/"
b64='ABCDEFGHIJKLMNOPORSTUVWXYZabcdefghiklmnopqrstuvwxyz0123456789+/
ciphertext= 'BInaEi=='
for ch in ciphertext:
    if (ch in tab):
        s+= b64[string.find(tab,str(ch))]
    elif(ch==='):
        S+='-1
print base64.decodestring(s)
```

具体的AES的解密算法为：

```
from Crypto.Cipher import AES
import binascii

raw = ' 37 f3 1f 04 51 20 e0 b5  86 ac b6 0f 65 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 ' ❶

ciphertext = binascii.unhexlify(raw.replace(' ','')) ❷
obj = AES.new('ijklmnopqrstuvwx', AES.MODE_CBC) ❸
print 'Plaintext is:\n' + obj.decrypt(ciphertext) ❹
```

习题解答

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？**

   动态分析可能找出的一些看似随机的加密内容，程序的输出中没有可以识别的字符串，所以没有什么东西暗示了使用加密。

2. **使用静态分析搜索字符 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？**

   搜索xor指令发现了6个可能与加密相关的单独函数，但是加密的类型一开始并不明显。

3. **使用静态工具，如 FindCvpt2、KANAL 以及DA 插件识别一些其他类型的加案机制。发现的结果与搜索字符XOR结果比较如何？**

   这三种技术都识别了高级加密标准AES算法（Rijndael算法），它与识别的6个XOR函数相关，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。

4. **恶意代码使用哪两种加密技术？**

   恶意代码使用AES和自定义的Base64加密

5. **对于每一种加密技术，它们的密钥是什么？**

   AES密钥是：ijklmnopqrstuvwx

   自定义加密的索引字符串是：
   CDEFGHIJKLMNOPQRSTUVWXYZABcdefghijklmnopqrstuvwxyzab0123456789+/

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？**

   对于自定义Base64加密的实现，索引字符串已经足够了，但是对于AES，实现解密可能需要密钥之外的变量，如果使用密钥生成算法，则包括密钥生成算法、密钥大小、操作模式，如果需要还包括向量的初始化等。

7. **恶意代码做了什么？**

   恶意代码使用以自定义Base64加密算法加密传入命令和以AES加密传出shell命令响应来建立反连命令shell。

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？**

   见上述具体分析