

**《恶意代码分析与防治技术》课程实验报告**

**实验13**

****

学 院 网络空间安全学院

专 业 信息安全

学 号 2112060

姓 名 孙蕗

1. **实验目的**

通过IDA Pro分析恶意代码，识别其中的字符串，查看其行为，以及是否存在加密机制。

使用静态分析工具和插件，如FindCrypt2、Krypto ANALyzer(KANAL)以及IDA熵插件，来识别恶意代码中可能存在的其他加密机制。

研究恶意代码的通信行为，包括恶意代码如何加密和发送数据。

通过监控恶意代码的运行行为，学习如何追踪其系统交互、文件操作和网络通信，以深入了解恶意代码的功能和攻击手段。

1. **实验原理**

利用IDA Pro查看恶意代码的字符串列表，注意到包含http的字符串，关注可能被加密的字符串。

使用IDA Pro搜索非归零xor指令，找到可能的加密操作，进行进一步分析。

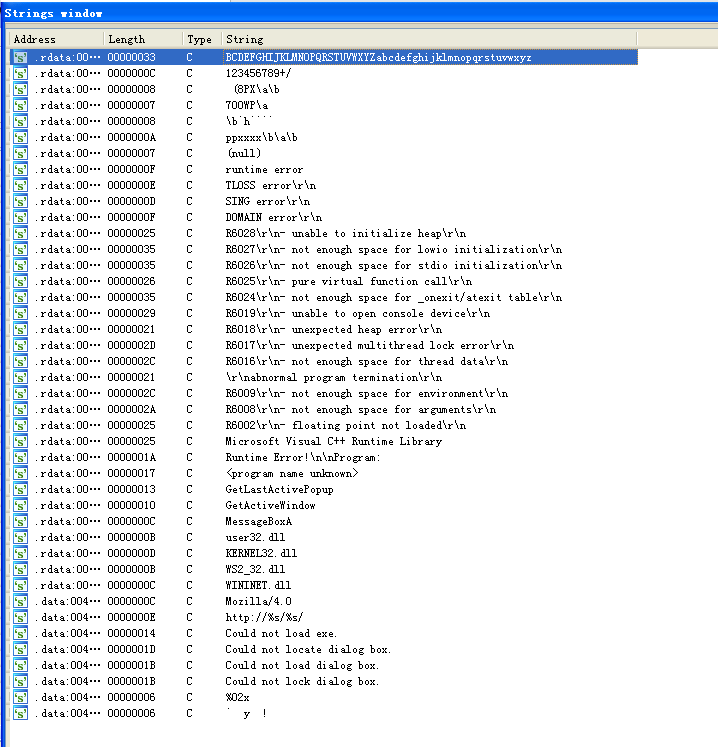
使用静态分析工具和插件，如FindCrypt2、Krypto ANALyzer(KANAL)以及IDA熵插件，来识别恶意代码中可能存在的其他加密机制。

分析恶意代码的通信行为，包括基于识别的加密机制来理解数据的传输方式。

在虚拟机环境中运行恶意代码，使用动态分析工具（如Process Monitor、Wireshark）监控其行为，包括文件系统活动、注册表修改、网络通信等，揭示加密数据和远程通信的细节。此外，通过分析动态行为，寻找与 C2（Command and Control）服务器的通信模式，以揭示攻击者可能采取的控制和操作手段。

1. **实验过程**
2. **Lab13-1**

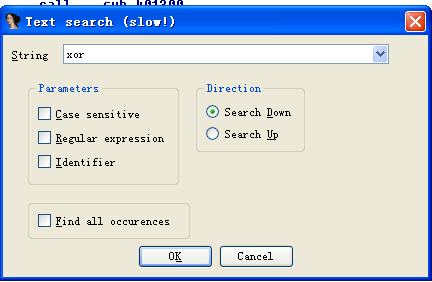
Ida pro打开Lab13-1，查看字符串列表

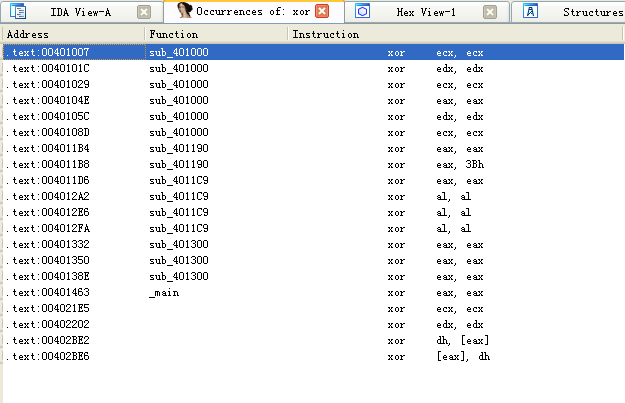


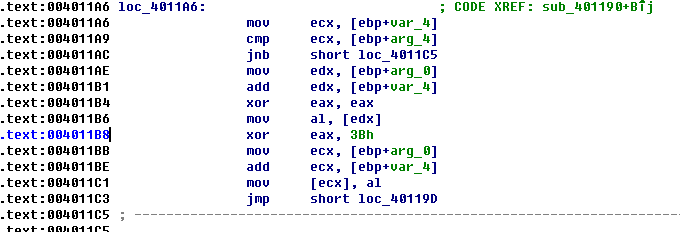
注意到有http字符串，并且%s表示有输出的值。

监视Lab13-01.exe的行为，注意到恶意代码访问www.practicalanalysis.com，看到Mozilla/4.0,但是没有发现字符串aG9zdG5hbWUtZm9v和www.practicalmalwareanalysis.com。因此，假设这些字符串可能被恶意代码加密。

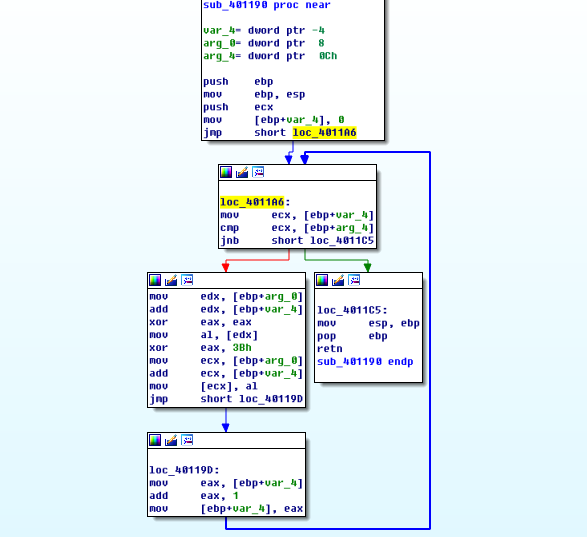
用IDA Pro搜索所有非归零xor指令，找到3个，但是其中两个(0x00402BE2和0x00402BE6处)是库代码。忽略这些代码，只留下sub\_401190中0x004011B8处的xor eax,3Bh指令。



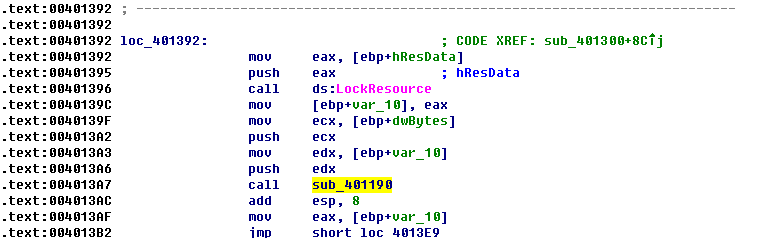




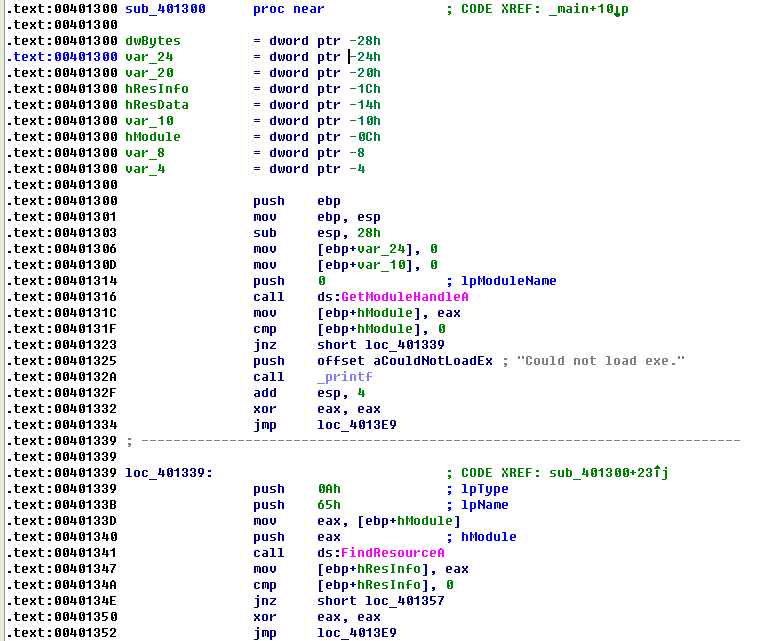
Loc\_4011A6包含一个小的循环，它递增一个计数器(var\_4),并且用0x3B与缓冲区(arg\_0)中原始内容的异或结果来修改缓冲区。其他被异或的参数(arg\_4)是缓冲区的长度。函数sub\_401190,用静态字节0x3B执行单字节XOR加密，并将缓冲区及其长度作为它的参数。

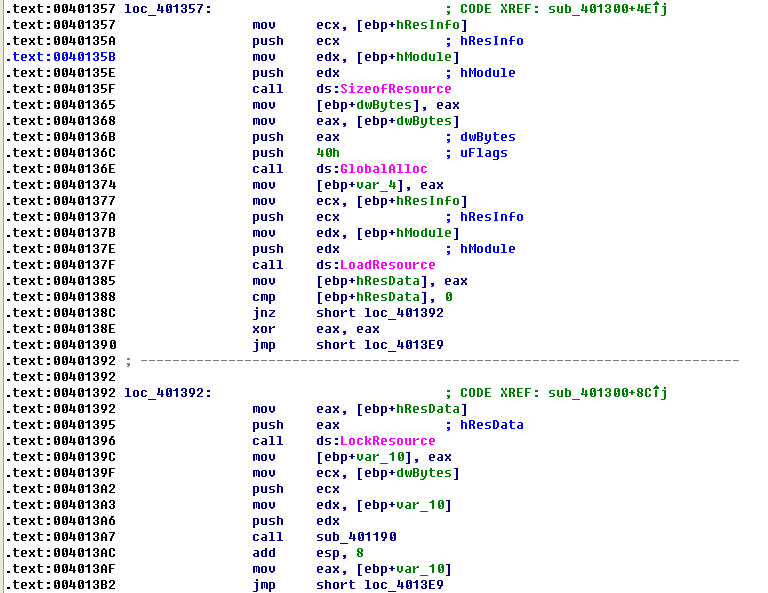


函数sub\_401300是调用sub\_401190的唯一函数。

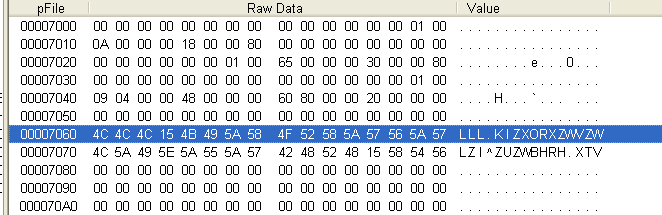


跟踪它调用sub\_401190之前的代码块，依次看到调用GetModuleHandleA、FindResourceA、SizeofResource、GlobalAlloc、LoadResource以及LockResource。调用sub\_401190之前，恶意代码会对资源做一些处理。

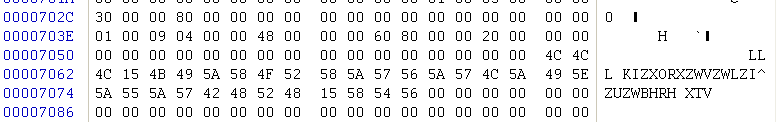




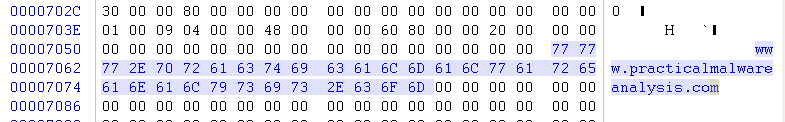
查看FindResourceA函数，IDA Pro为标注了它的参数。其中1pType是0xA,它表示资源数据是应用程序预定义的还是原始数据。1pName既可以是一个名字，也可以是一个索引号。本例中，它是一个索引号。由于函数引用了ID为101的资源，使用PEview查看PE文件中的资源，并且查找索引号为101(0x65)的RCDATA资源，资源在偏移量0x7060处，长度是32字节。



在WinHex中打开可执行文件时，从7060至7080都高亮显示。然后选择Edit→Modify Data,选择XOR,再输入3B。





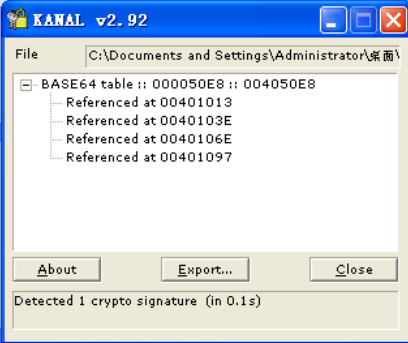


上半部分显示了数据的原始版本，下半部分显示了用0x3B对每个字节异或之后的结果。上图清晰地显示了以加密形式存储的字符串www.praticalmalwareanalysis.com。

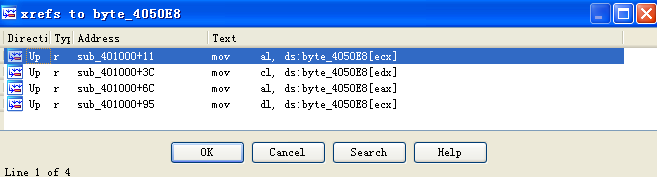
其中有两个字符串让我们怀疑可能被加密，我们发现了域名，但是没有发现GET请求字符串(本例中是aG9zdG5hbwUtZm9v)。为了找到GET字符串，使用PEiD的KANAL插件，它在0x004050E8处找到一个Base64编码表



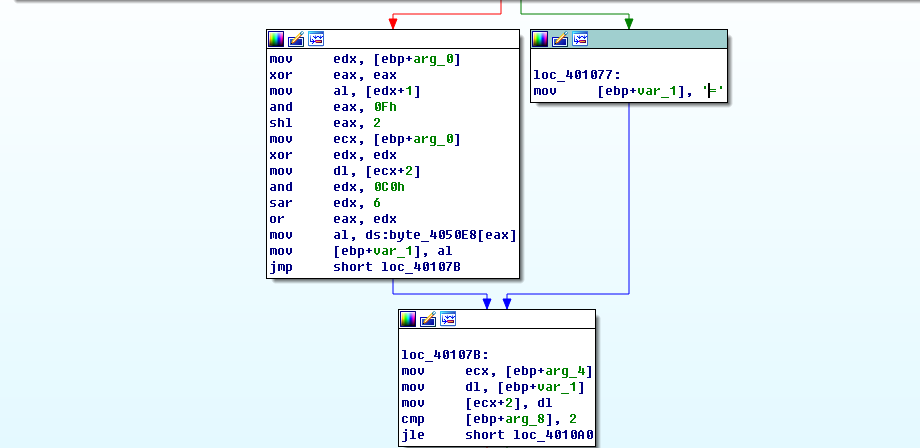
使用PEiD的Krypo ANALyzer插件，发现该恶意代码有一个BASE64 table，地址是0x004050E8，于是在 IDA中跳转到该位置，看到这里存着的是字符串 ABCDEFGHIJKLMNOPQRSTUVXYZabcdefghijklmnopqrstuvwxyz0123456789+/



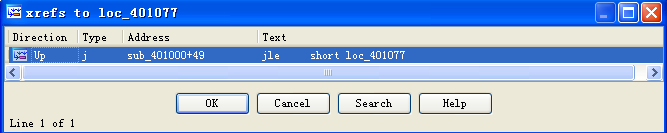
移动光标到Base64编码表，看到它是一个标准的Base64字符串：ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/。在IDA Pro的sub\_401000中这个字符串有4个交叉引用，它们位于以0x00401000开始的函数中，可将这个函数称为base64index。



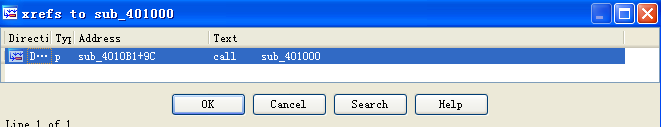
该函数中有一处对=的引用，这是Base64加密中的填充字符，因此该函数执行加密操作，调用该函数的函数是Base64加密函数。这里一共进行了三次的清空寄存器和存放的操作，这里是在对"GET"字样进行解密了。

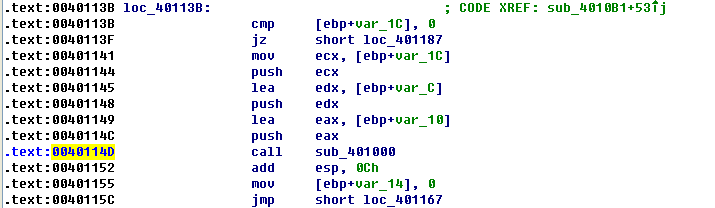


Base64编码在sub\_401000（base64index）被引用

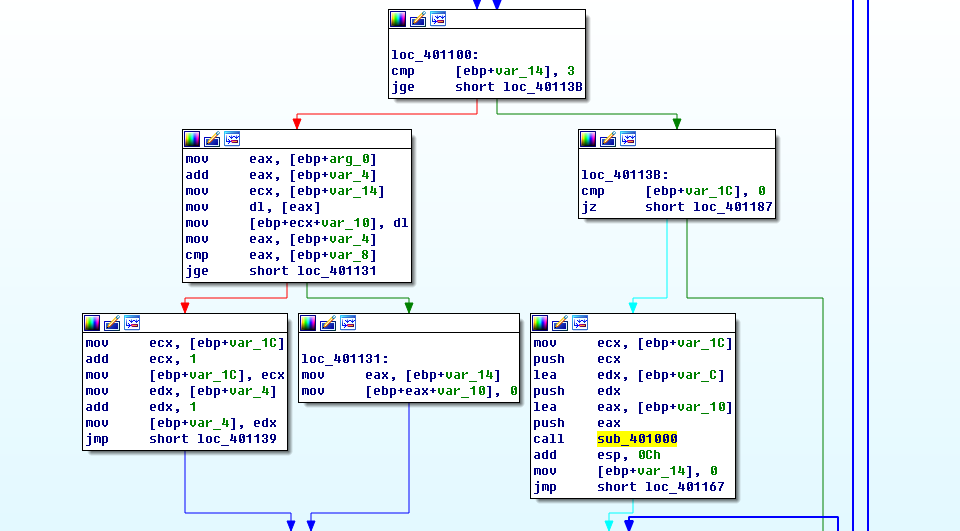


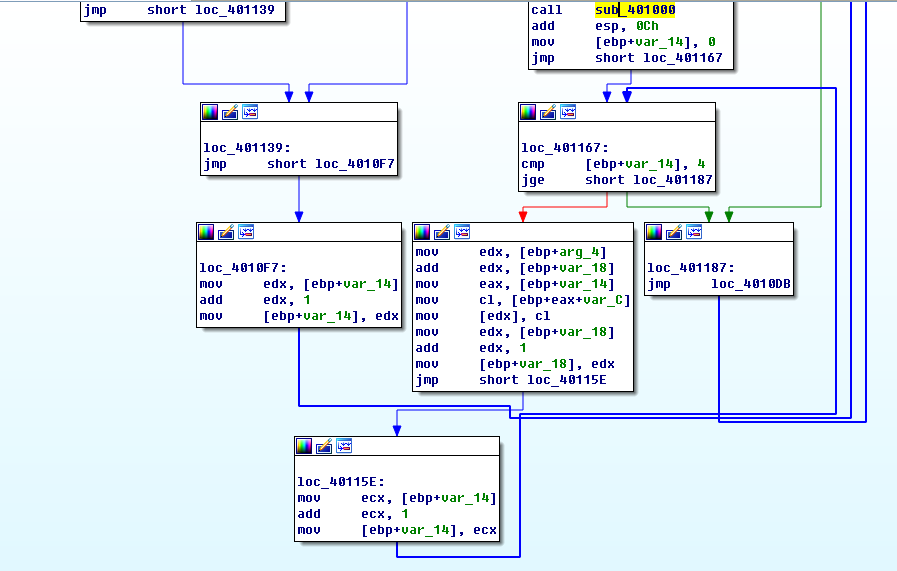
调用base64index的函数(位于0x004010B1)是真正的Base64加密函数。





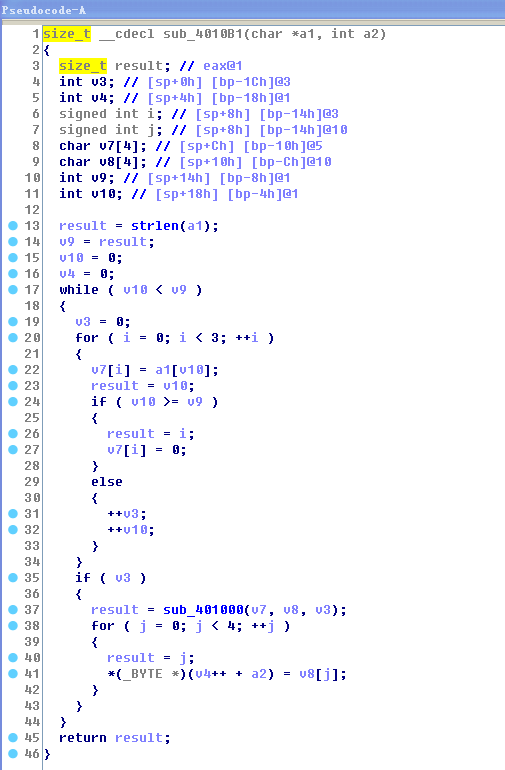
它的目的是将源字符串划分成3个字节的块，并且将3个字节块传递给base64index,从而将传入的3个字节加密成4个字节。





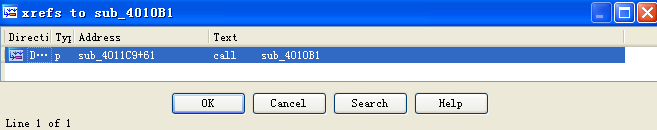
函数开头使用strlen查找源字符串的长度，在外部循环(代码块1oc\_401100)的开头与3比较(cmp [ebp+var\_14],3),当base64index返回结果后，在内部写循环的开头与4比较(cmp [ebp+var\_14],4)。

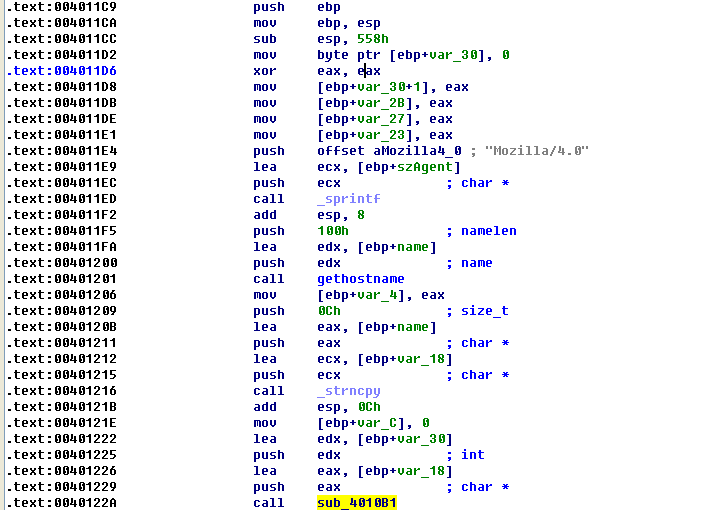
base64\_encode就是Base64编码的主函数，它用源字符和Base64加密转换的目的缓冲区作为参数。

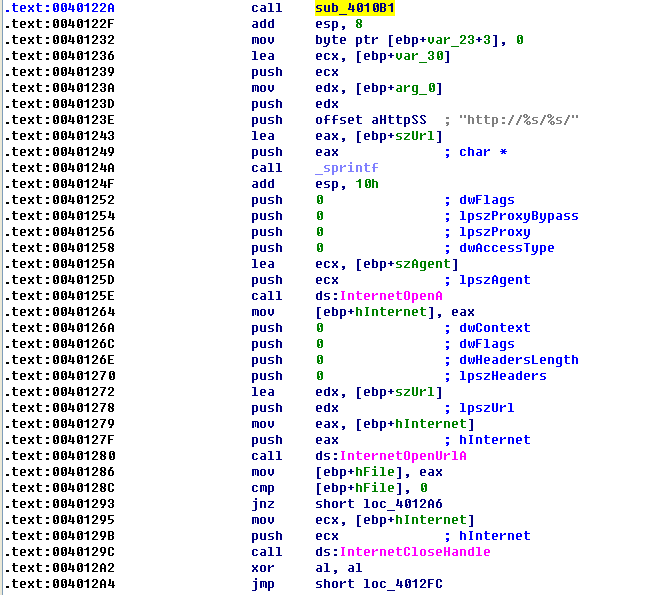


F5将sub\_4010B1转为C语言，看见这个循环的判定条件是v10和v9的大小，并且v10的初始值是0，在之后的循环体内部v10会执行一个自增的操作，那么这个v10其实也就是相当于一个下角标的作用，来控制循环次数。并且v9字符串的初始值设置的是 strlen(a1) ，也就是a1字符串的长度，那么也就是说这里循环就是遍历了整个字符串。循环体内部每次取出来三个字符，然后利用base64进行解密的操作。如果主机名小于12个字节并且不能被3整除，则可能使用填充字符。

检查对base64\_encode（0x004000B1的交叉引用），位于0x004011C9处的函数中。





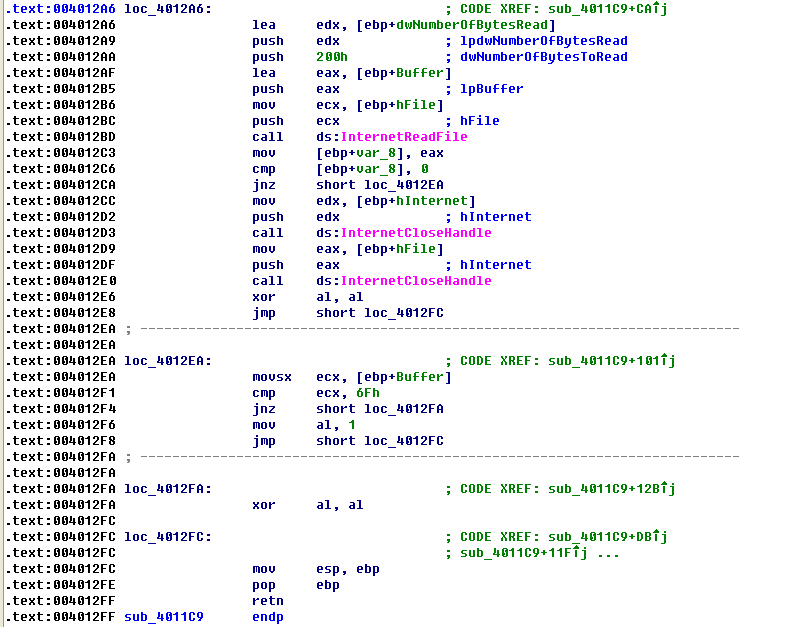


查看传递到函数base64\_encode的目的字符串，作为sprintf的第4个参数入栈(0x00401236处所示)。具体说，以http://%s/%s/(0x0040123E处)格式的第2个字符串是URL的路径。这与前面识别的字符串aG9zdG5hbwUtZm9v一致。

跟踪传递给函数base64\_encode的源字符串，可以看到，它是0x00401216处strncpy函数的输出。并且strncpy的输入是0x00401201处gethostname的输出。因此，可以明白加密的URL路径源字符串是主机名。strncpy函数仅复制了主机名的前12个字节，如0x00401209处所示。

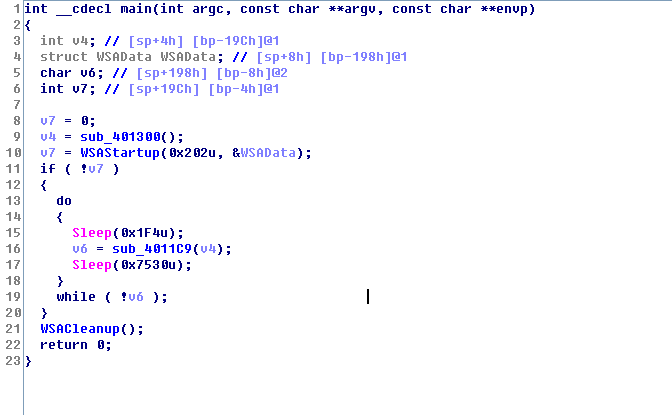
用Base64加密字符串表示的主机名永远不能超过16个宇符，因为12字符×4/3的Base64加密扩展等于16。另外，仍然有可能看到字符‘='填充到字符串末尾，但是仅当主机名小于12个字符且主机名的长度不能被3完全整除时，这种情况才会发生。

查看beacon中的剩余代码，看到它使用了WinINet(InternetopenA、InternetOpenUrlA和InternetReadFile)打开并且读取URL。返回数据的第一个字符与字母o比较。如果第一个字母是o,则beacon返回1,否则返回0。



主函数由一个循环、Sleep调用以及beacon调用组成。当beacon(0x004011C9)返回真时(获得以o开始的Web响应),则循环退出程序终止。





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出一个特定的通信信号，当接收到一个特定的回应后，则终止。

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。

1. **使用IDAPro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函数中一个单字节XOR加密循环的指令。

1. **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。

1. **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?**

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+/

1. **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。

1. **Base64编码函数在反汇编的何处?**

Base64加密函数从0x004010B1处开始。

1. **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。

1. **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。

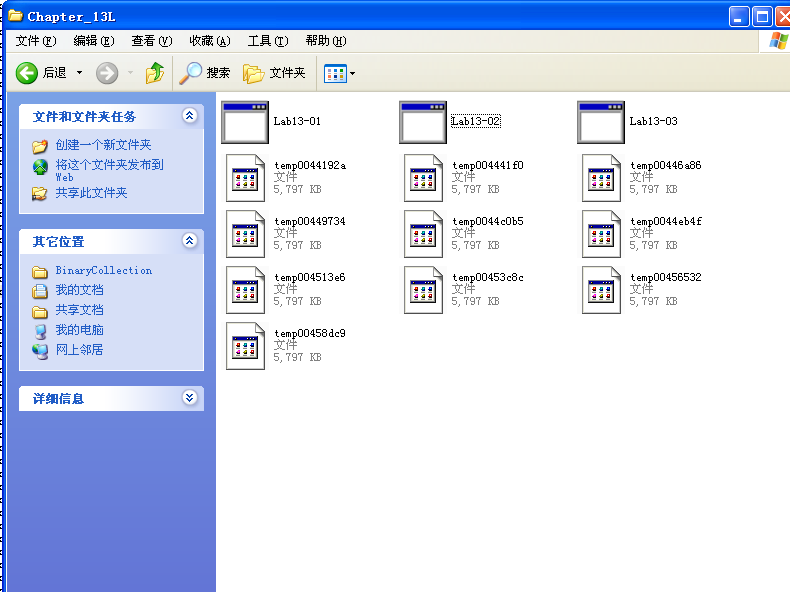
1. **这个恶意代码做了什么?**

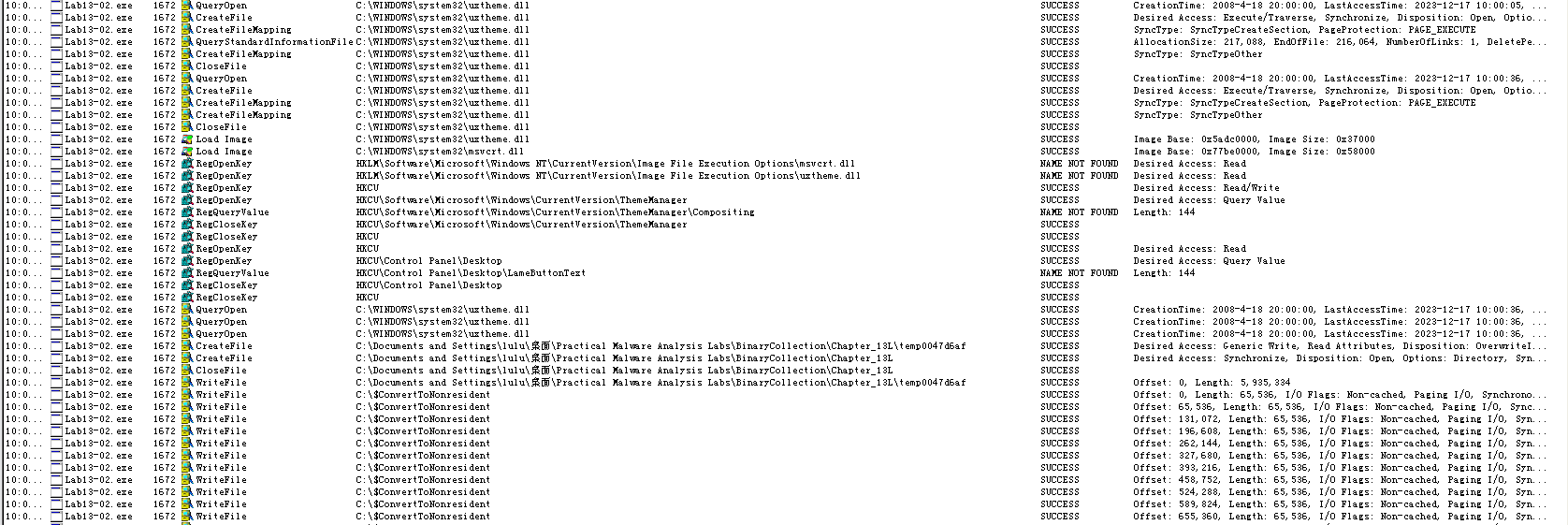
Lab13-01.exe用加密的主机名发送一个特定信号，直到接收特定的回应后退出。

1. **Lab13-2**

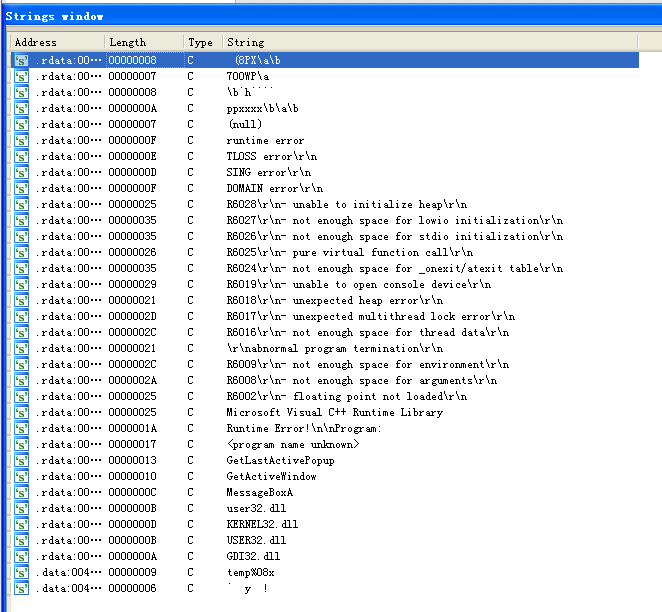
启动恶意代码，看到它在固定的时间间隔内在恶意代码的当前目录下创建了一些新文件。这些文件相当大(几兆大小)并且文件似乎包含一些随机的数据，文件名以temp开始并且以一些看似随机的字符结束。

使用Procmon进行监控，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 5797KB。

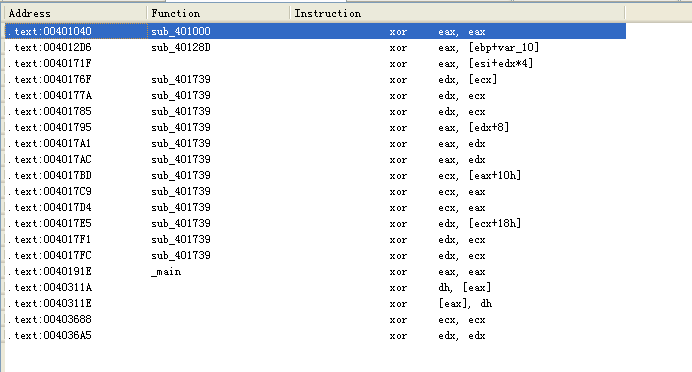




查看Lab13-02.exe的字符串列表



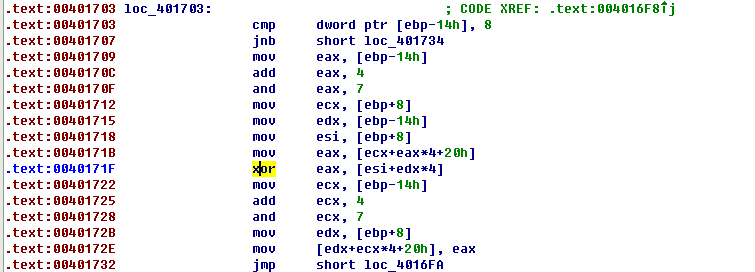
使用静态分析搜索恶意代码中的加密证据。PEiD KANAL插件，IDA Pro的FindCrypt2插件以及IDA熵插件都没有发现任何有价值的东西。在IDA中使用搜索功能查找xor。

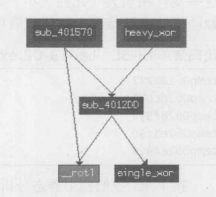


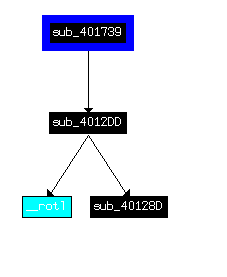
可以看见有很多条的xor指令，并且其中大部分来自sub\_401739，还有一个条来自sub\_40128D。忽略其中用于将寄存器清零的和库函数中的操作，其余的xor指令集中在函数sub\_40128D和sub\_401739中，并且搜索结果中的第三条指令地址0x0040171F在代码块中没有对应的函数。

因为sub\_401739拥有如此多的xor指令，将sub\_401739称为heavy\_xor。而sub\_40128D拥有一条xor指令，将sub\_40128D称为single\_xor。heavy\_xor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，发现single\_xor与heavy\_xor相关，因为single\_xor的调用者也被heavy\_xor调用。

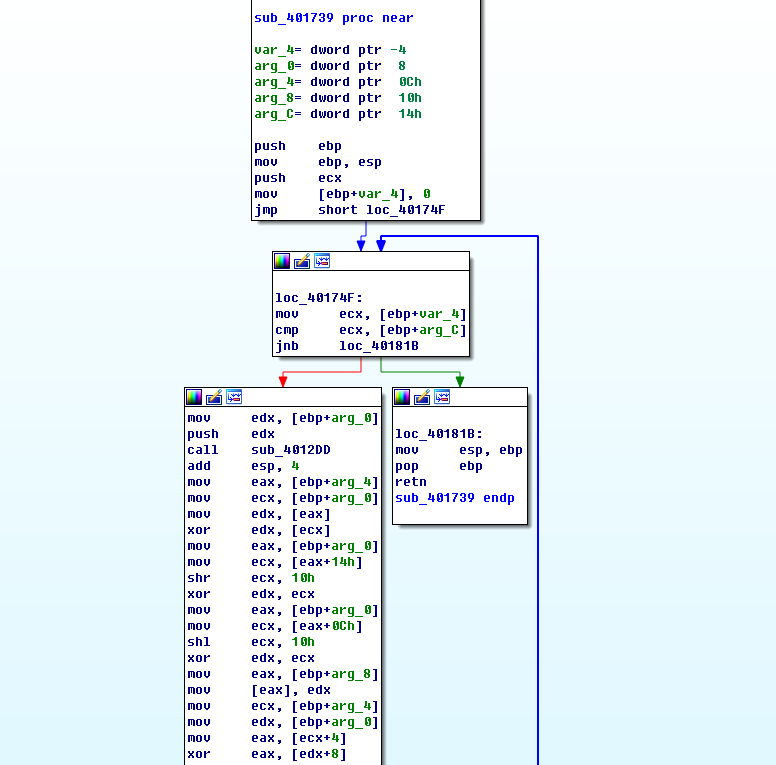
查看0x0040171F的xor指令，看到它位于一个函数中，但是由于很少使用，这个函数并没有被自动识别。在0x00401570定义函数导致创建包含了前面孤立的xor指令函数。这个未使用的函数也与同组内可能的加密函数有关联。



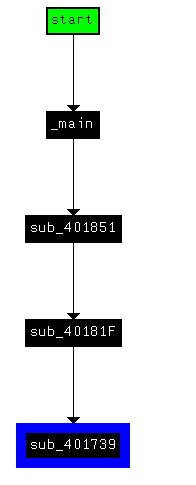


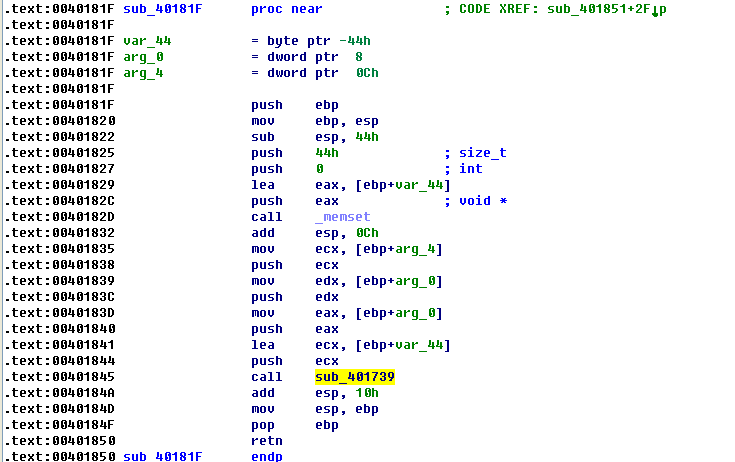


查看拥有最多xor指令的函数sub\_401739，它包含一个循环，循环体中大多是xor、SHL和SHR指令，只有一次函数调用，调用的是sub\_4012DD，而这个函数又会调用sub\_40128D，即拥有xor指令的另一个函数，推测这两处xor指令相关。



为了证实heavy\_xor是一个加密函数，看一下它与写入到磁盘上temp文件有何种关系。找到了数据写入到磁盘的位置，然后向前回溯确定怎么使用的加密函数。查看导入函数，看到了WriteFile。



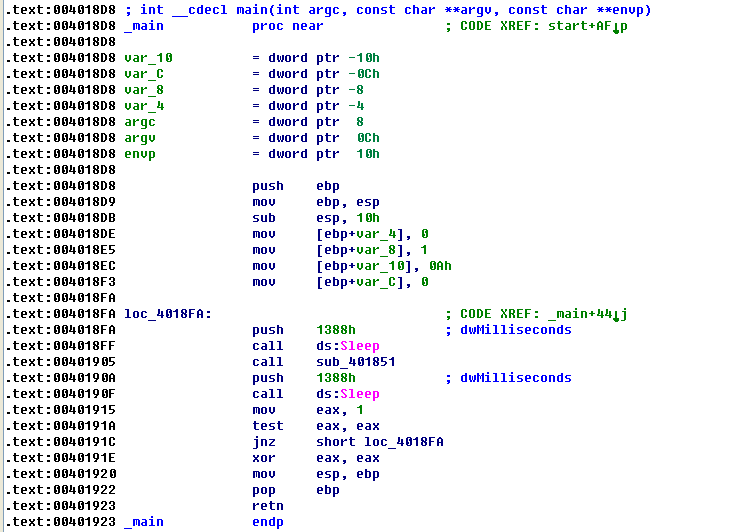


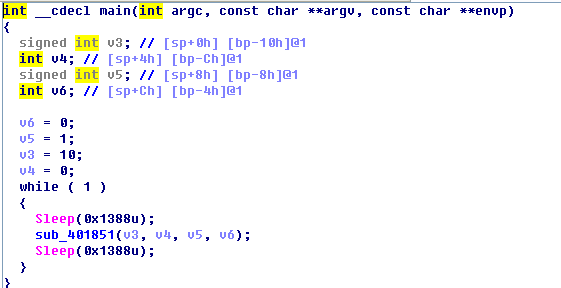
Sub\_40181F只是一个heavy\_xor函数的封装器。



查看sub\_401739的调用关系图，猜测sub\_401851与加密有关。

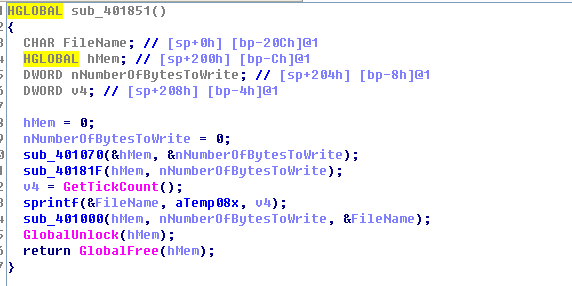
查看main函数，





call指令调用了sub\_401851。

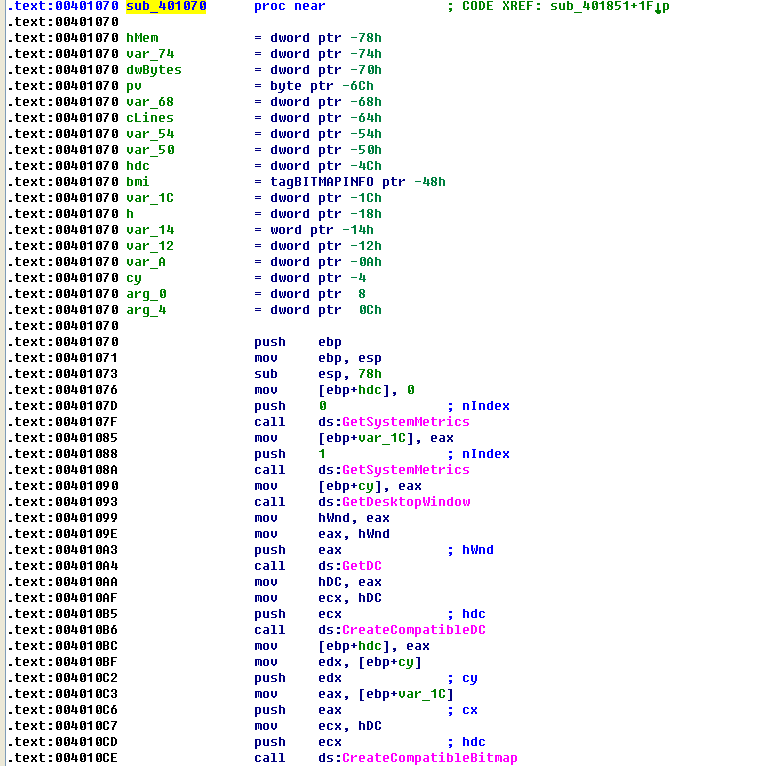


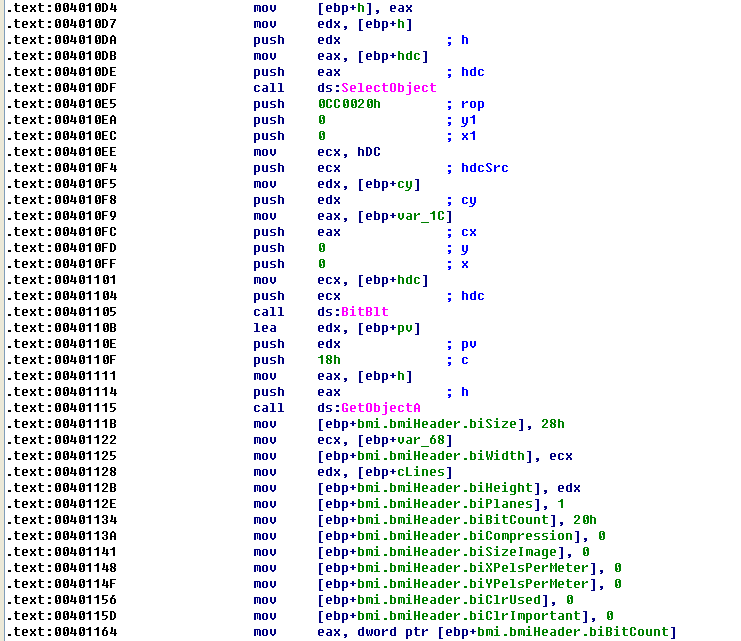


发现这个函数调用了几个API，而且还有sub\_401070和sub\_40181F。

这里出现了和生成的文件名有关的部分，显示用了个GetTickCount函数返回从操作系统启动所经过的毫秒数，将其传递下来作为后面函数的参数，还调用了sub\_401000函数。GetTickCount结果的格式化字符串“temp%08x”发现了原文件名，它是当前时间的十六进制打印。在0x004018A9处标注了文件名，猜想sub\_401070用来获取内容（重命名为getContent）,sub\_40181F用来加密内容（重命名为encodingWrapper）。

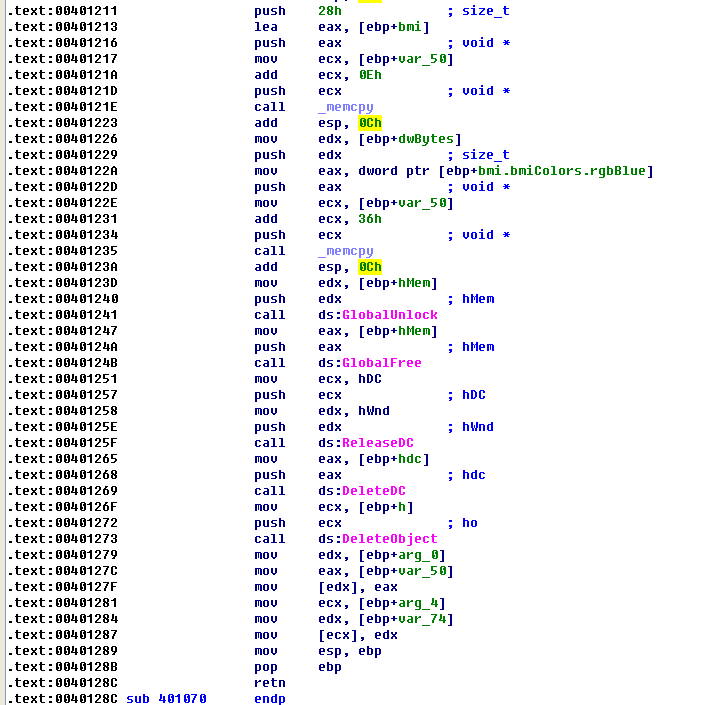
查看sub\_401070

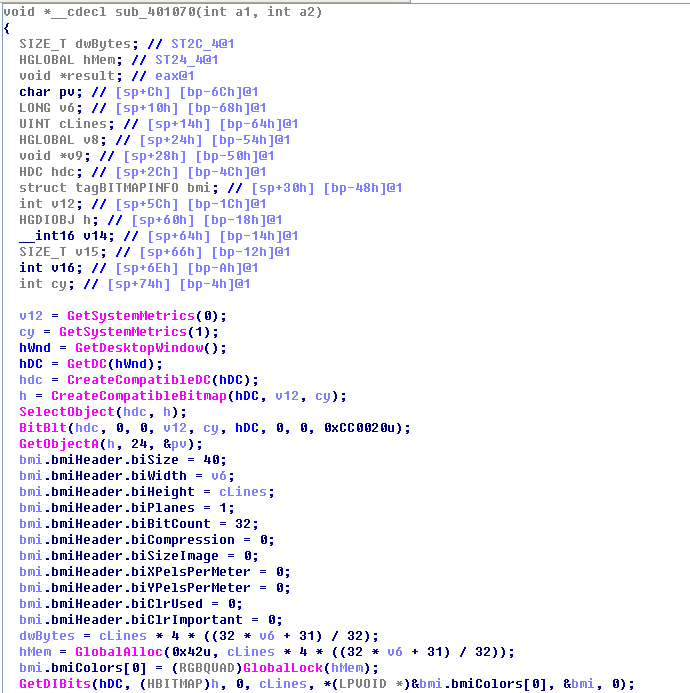


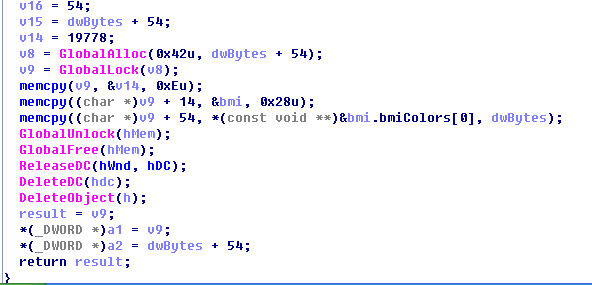








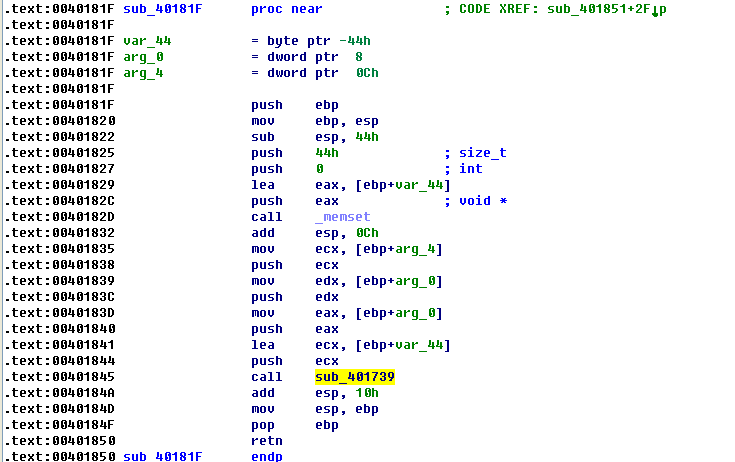




看到sub\_401070是截屏抓屏的功能。完整分析过后可知程序调用了获取用户的桌面，创建bitmap对象，加密放在桌面。GetDesktopWindow获取覆盖整个屏幕桌面窗口的一个句柄，函数BitBlt和GetDIBits获取位图信息并将它们复制到缓冲区。

因此分析得知恶意代码反复抓取用户的桌面截图并且将加密版本的抓屏信息写入到一个文件。

查看sub\_40181F



看到sub\_40181F仅仅是一个heavy\_xor函数的封装器。函数encodingWrapper首先设置了4个加密的参数：一个使用之前清空的本地变量，从doStuffAndWriteFile传递过来指向同一个缓冲区的两个指针以及一个从doStuffAndWriteFile传递过来的缓冲区大小。两个指针指向同一个内存缓冲区暗示编码函数携带源和目的缓冲区以及一个长度，这种情况下，加密将在适当的地方执行。

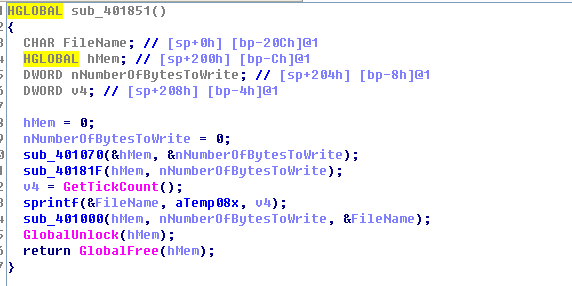
问题1中的特征发现是创建了很多具有一定特征的文件，而创建文件要使用的就是WriteFile函数。检查WriteFile的交叉引用，找到了sub\_401000,它用一个缓冲区，一个缓冲区长度，一个文件名作为参数，它打开一个文件并且将缓冲区数据写入到这个文件。



将sub\_401000重命名为writeBufferToFile。sub\_401851是调用writeBufferToFile的唯一函数。

查看sub\_401851,将其命名为doStuffAndwriteFile，作为0x004018B8处调用writeBufferToFile函数前的准备。



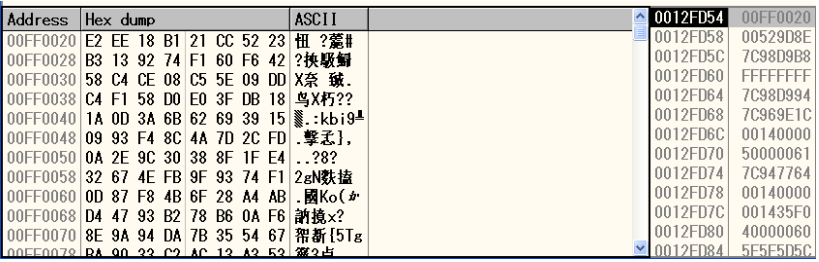


发现这个函数调用了几个API，而且还有sub\_401070、sub\_40181F和sub\_401000。

利用抓取到的一个文件，通过加密算法逆向运行它，可以提取到原始抓屏图片(假设这种算法使用一种流密码并且加密过程是可逆的，即加密和解密完成相同的事情)。因为对这种算法的使用有了一些线索，其实最容易实现的方法是用解密规范让代码为我们执行解密。

由于代码已经有指令获取缓冲区，加密它，并写入到文件，因此，以如下方式重新使用它们：执行加密前，让程序正常运行；使用希望解密的已保存到文件的缓冲区覆盖存放抓屏信息的缓冲区；让程序将输出结果写入到一个临时文件，它的文件名基于当前时间；写入第一个文件后，中断程序的运行。可以使用OllyDbg,或者能够提供更多灵活性的脚本，手动实现这种策略。

可以使用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。



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。（假设加密和解密用的是同样的函数）

确保文件大小与加密函数传入的第2个参数一致。如果在恶意代码开始运行与运行解密之间没有改变屏幕的分辨率，则大小应该一致。如果文件长度大于内存缓冲区的长度，则这种技术可能会失败。

用更通用的方法(以一种不依赖可用缓冲区大小的方式)实现这种规范的解密策略，在Immunity Debug调试器中使用了基于Python的调试器API。创建一个Python脚本，将它以扩展名.,py保存在ImmDbg安装目录下的Py;Scripts文件夹中。



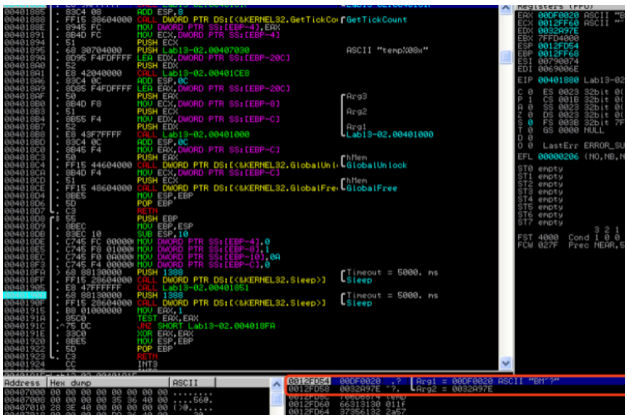
第一个断点刚好在参数被压入堆栈前中断。open调用打开一个已经写入到文件系统的加密文件。随后几行将文件读入内存并且计算缓冲区的大小。remoteVirtualAlloc调用用来在正在执行的进程内存空间中创建一个大小合适的缓冲区。另外，writeMemory用来将文件的内容复制到这个新创建的缓冲区中，两个writeLong调用替换堆栈中加密的缓冲区和它大小的变量。随后几条指令则是将这些变量压入堆栈，供随后的加密例程和写文件使用。

在ImmDbg中打开恶意代码，选择ImmLib→Run Python Script,然后选择创建的脚本，则脚本将会执行且调试器会在第二个断点处中断。此时，恶意代码应该已经在它的当前目录中写了一个文件。进入恶意代码所在目录并且确定它最近写入的文件，将文件的扩展名修改为.bmp并且打开它。看到解密后的截图是恶意代码之前的抓屏。

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0x00401880），然后在写入文件之后的位置再下一个断点（位置为：0x00401905）。

想法是在到达加密的位置时，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去，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。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。发现就是刚刚电脑状态的截图，解密成功。



1. **使用动态分析，确定恶意代码创建了什么?**

Lab13-02.exe在当前目录下创建一些较大且看似随机的文件，这些文件的命名以temp开始，以不同的8个十六进制数字结束。

1. **使用静态分析技术，例如xor指令搜索、FindCrypt2、KANAL以及IDA熵插件，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种推荐的技术并没有发现什么。

1. **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?**

WirteFile调用之前可能会发现加密函数。

1. **加密函数在反汇编的何处?**

加密函数是sub\_40181F。

1. **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。

1. **你是否能够找到加密算法?如果没有，你如何解密这些内容?**

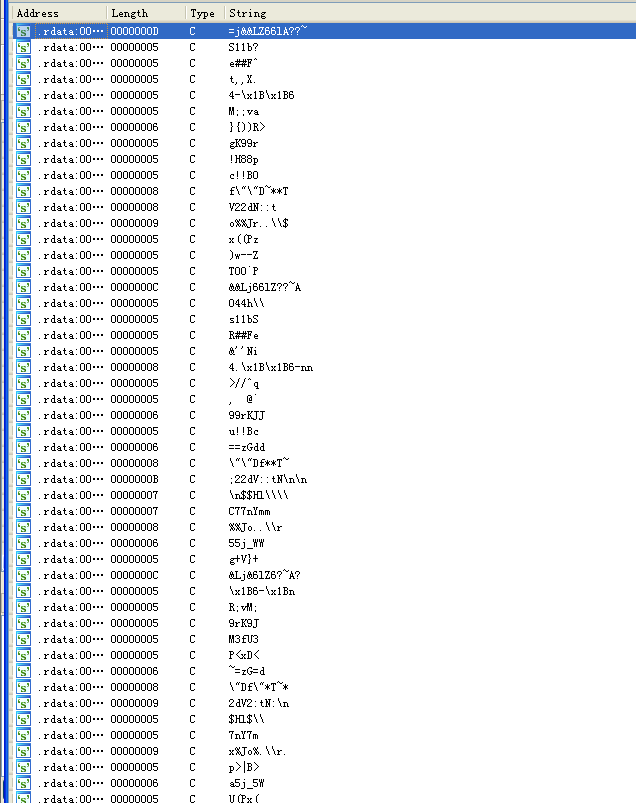
加密算法是不标准算法，并且不容易识别，最简单方法是通过解密工具解密流量。

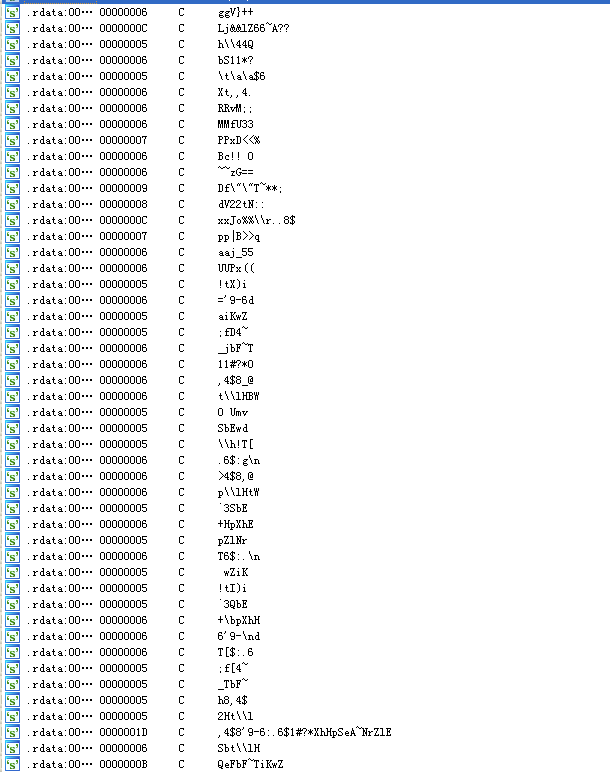
1. **使用解密工具，你是否能够恢复加密文件中的一个文件到原始文件?**

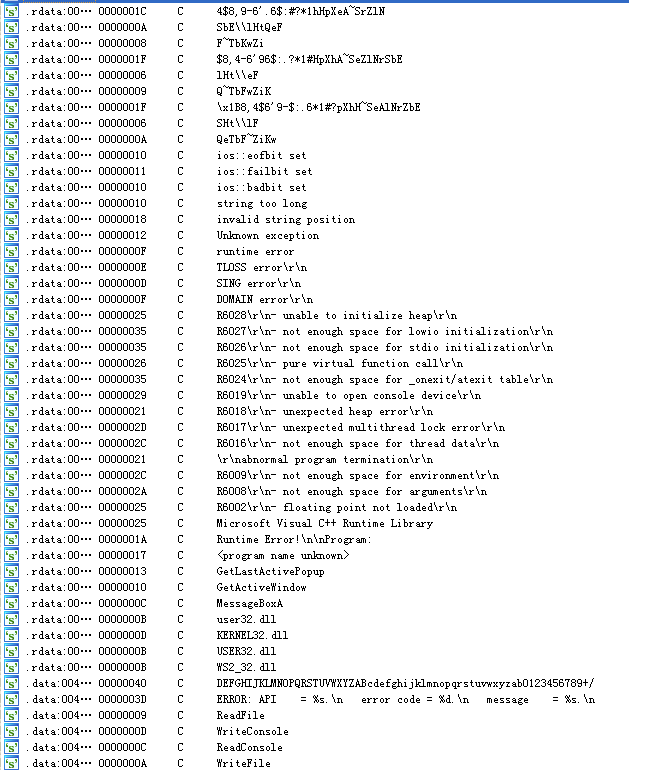
请查看详细分析过程。

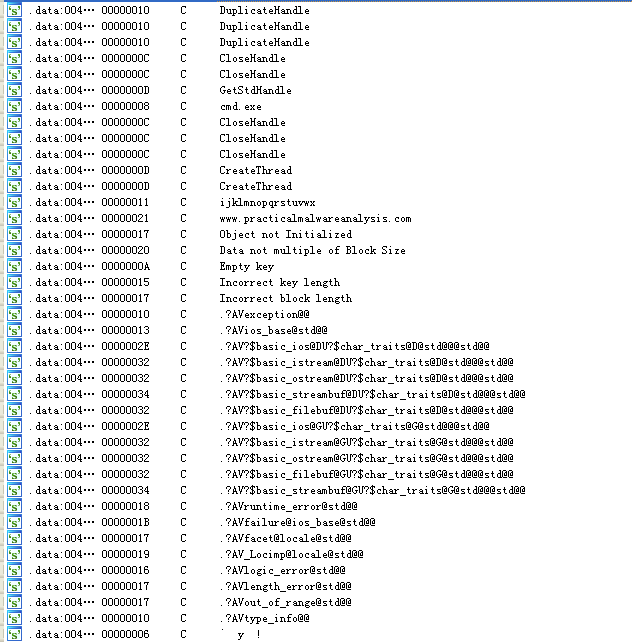
1. **Lab13-3**

查看Lab13-03.exe字符串



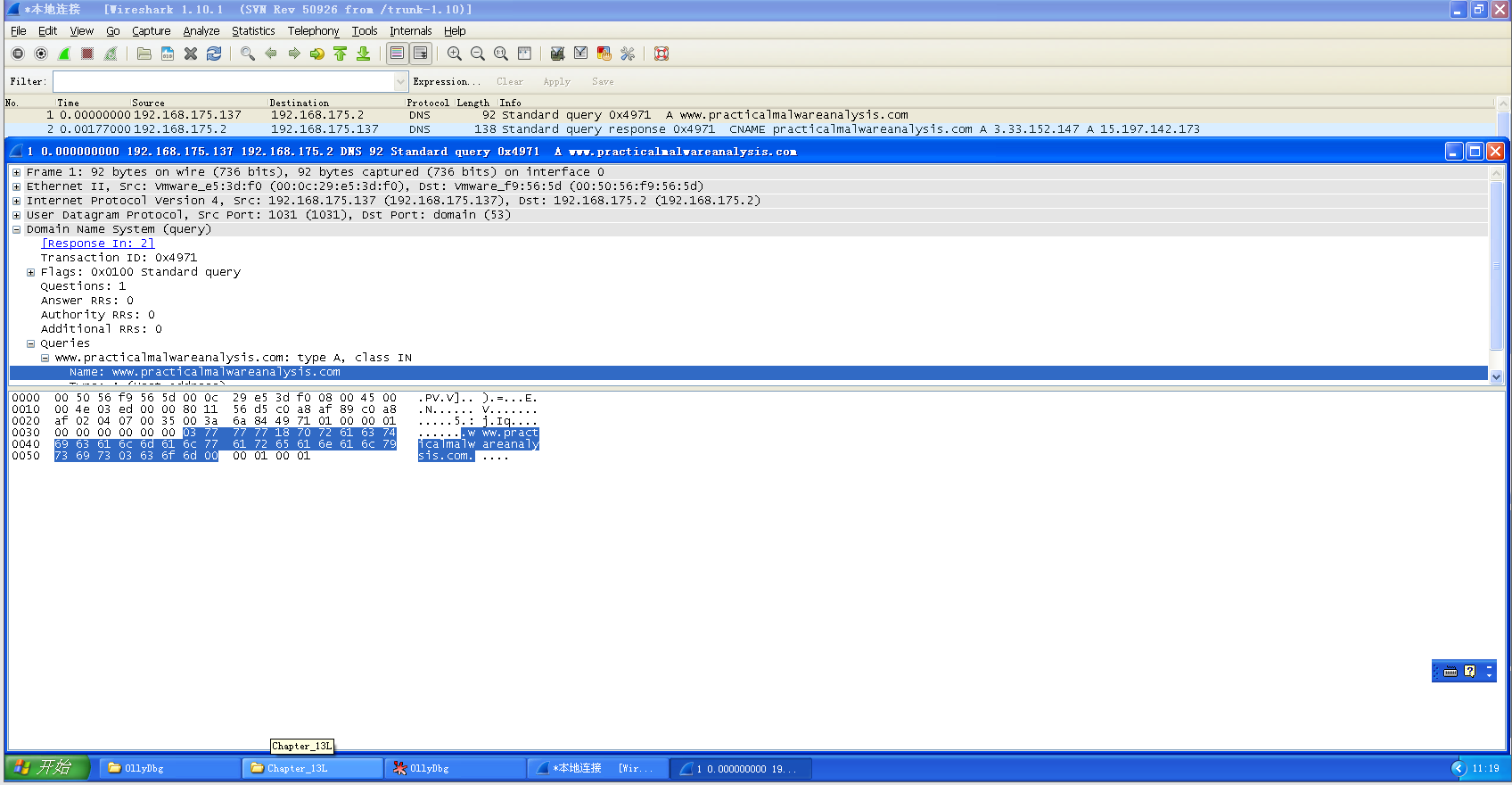






看到域名www.practicalmalwareanalysis.com，一些API函数名与一些乱码，可以认为是经过了加密，但是具体是哪些元素中使用了加密难以判断。有一个类似于base64加密的字符串（CDEFGHIJKLMNOPQRSTUVWXYZABcdefghijklmnopqrstuvwxyzab0123456789+/），发现下面的ERROR后面有一个=，还有格式化字符串，猜测会对这里进行一个加密。其他相关的字符串，例如WriteConsole和DuplicateHandle,它们可能是错误信息的一部分，类似于前面ReadConsole的错误信息。虽然不能知道加密了什么，但是在动态分析过程中看到的随机内容暗示了可能使用了加密。一些字符串暗示了恶意代码执行了加密，包括数据中没有块大小、空密钥、不正确的密钥长度及不正确的块大小。

使用wireshark进行分析



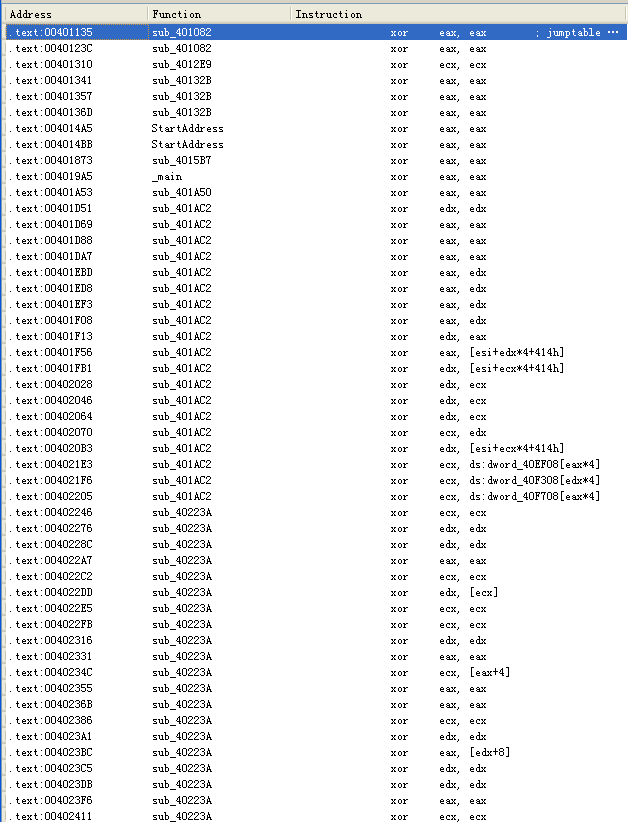
可以看见出现了刚刚分析得到的url。看到恶意代码尝试解析域名www.practicalmalwareanalysis.com，还注意到存在TCP的三次握手与四次挥手相关的操作，可以认为它在尝试连接远程主机。

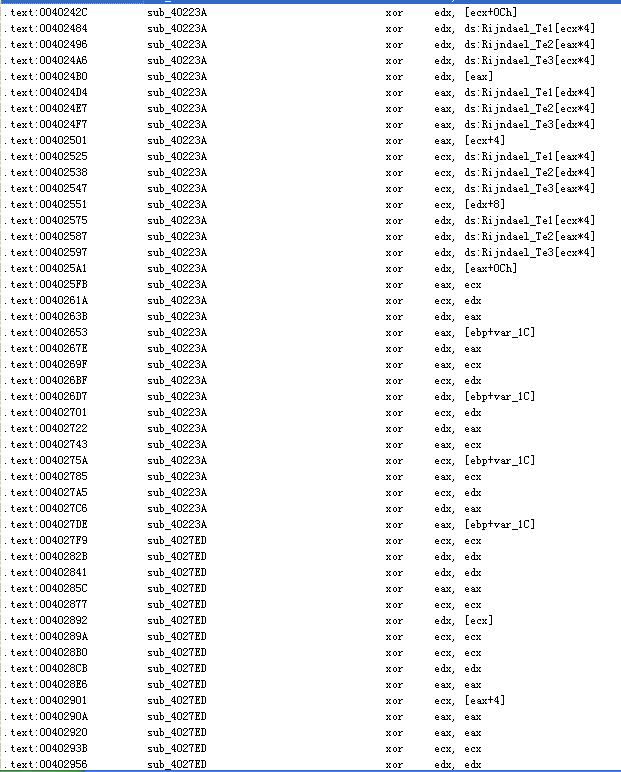
如果使用Netcat向这个连接发送内容，可以看到恶意代码用一些随机内容作为 响应，但是看起来都是不可读的乱码（可能被加密了），如果从Netcat端终止这个连接，可以看到：

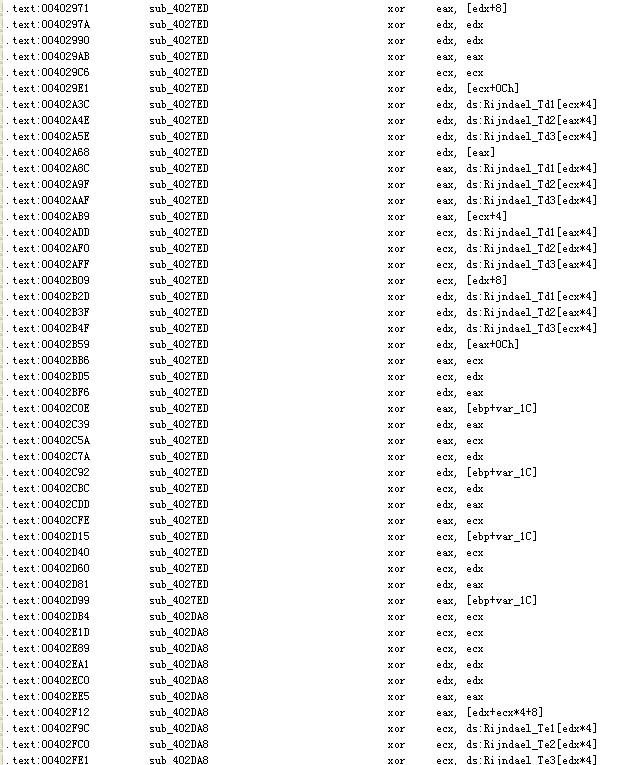
|  |
| --- |
| Go ERROR: API = ReadConsole.  error Code = 0.  message = The operation completed successfully. |

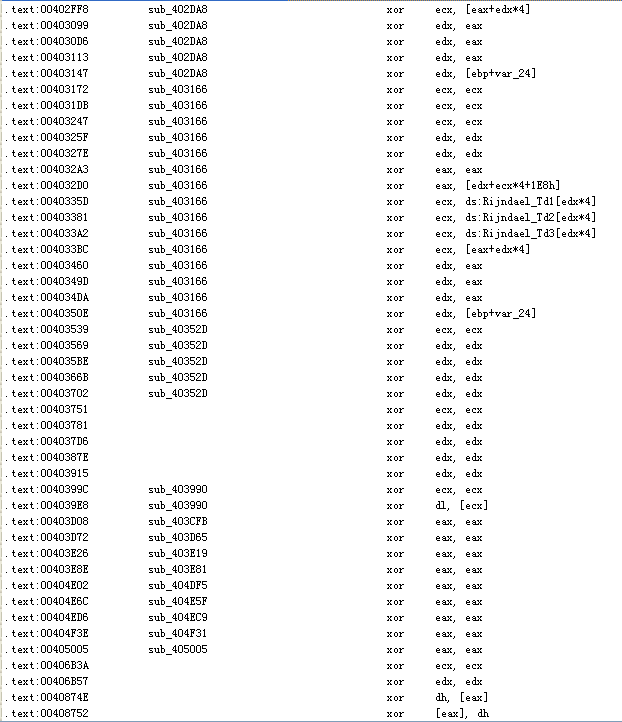
与输出字符串进行对比，看到了域名和看起来像前面提到的出错信息的字符串。

IDA中搜索xor

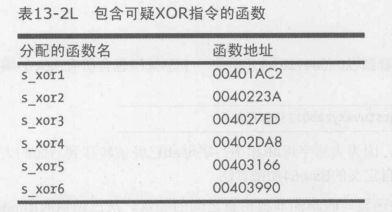




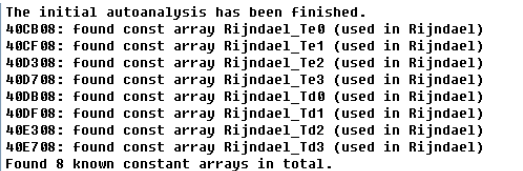




结果中有大量的xor，需要排除用于寄存器清零和库函数相关的xor指令。所以最后可以得到的是发现了6个可能与加密相关的单独函数，但是加密的类型看不出来。鉴于大量识别出的函数，标注它们并且看一下它们与将要采用的附加技术有何关联。

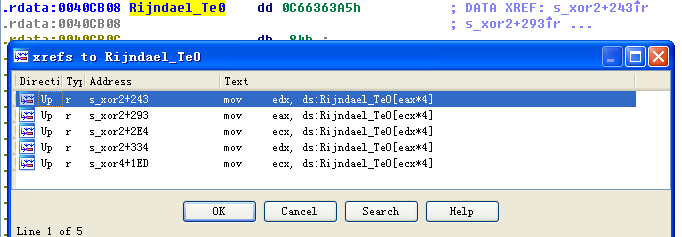


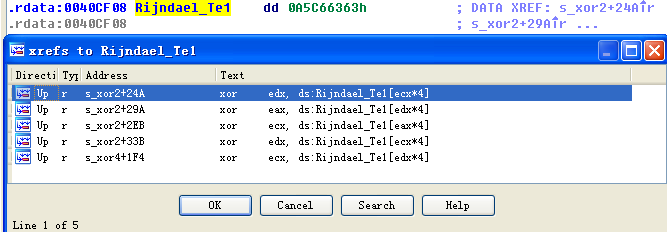
用IDA Pro的FindCrypt2插件

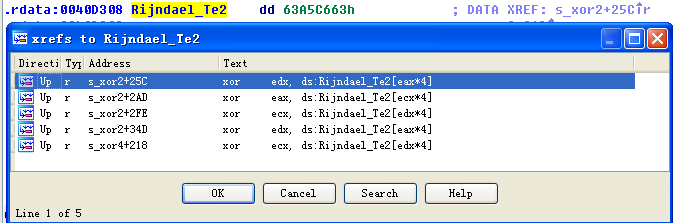


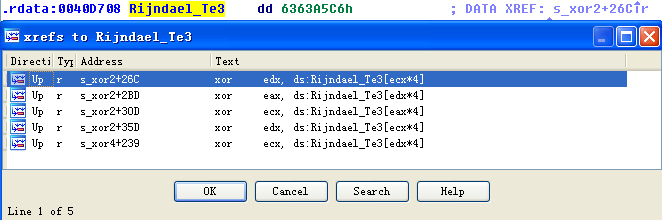
可以看到是Rijndeal算法，是AES的曾用名

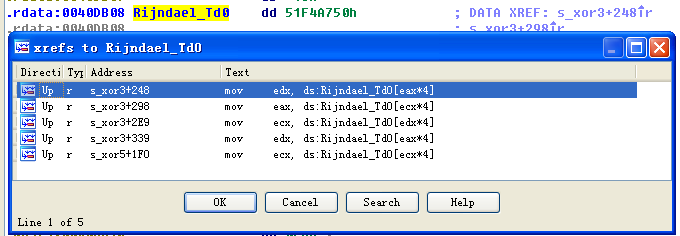
查看交叉引用，s\_xor2,s\_xor4通过加密常量Te0、Te1、Te2、Te3关联，s\_xor3和s\_xor5通过解密常量Td0、Td1、Td2、Td3、Td4关联。s\_xor2,s\_xor4和AES加密相关，而s\_xor3,x\_xor5与AES解密相关。

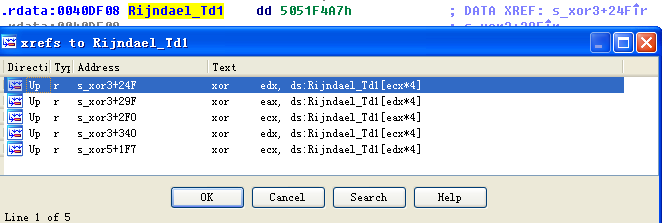


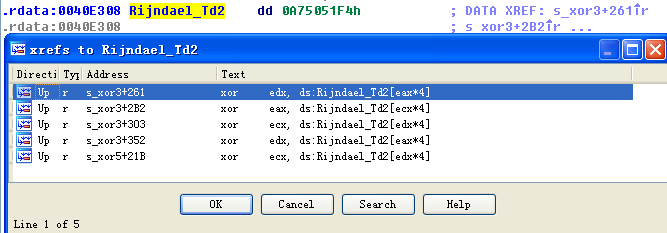


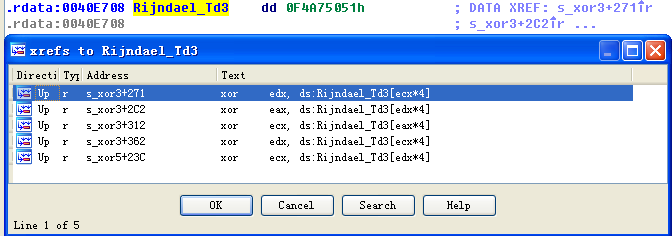












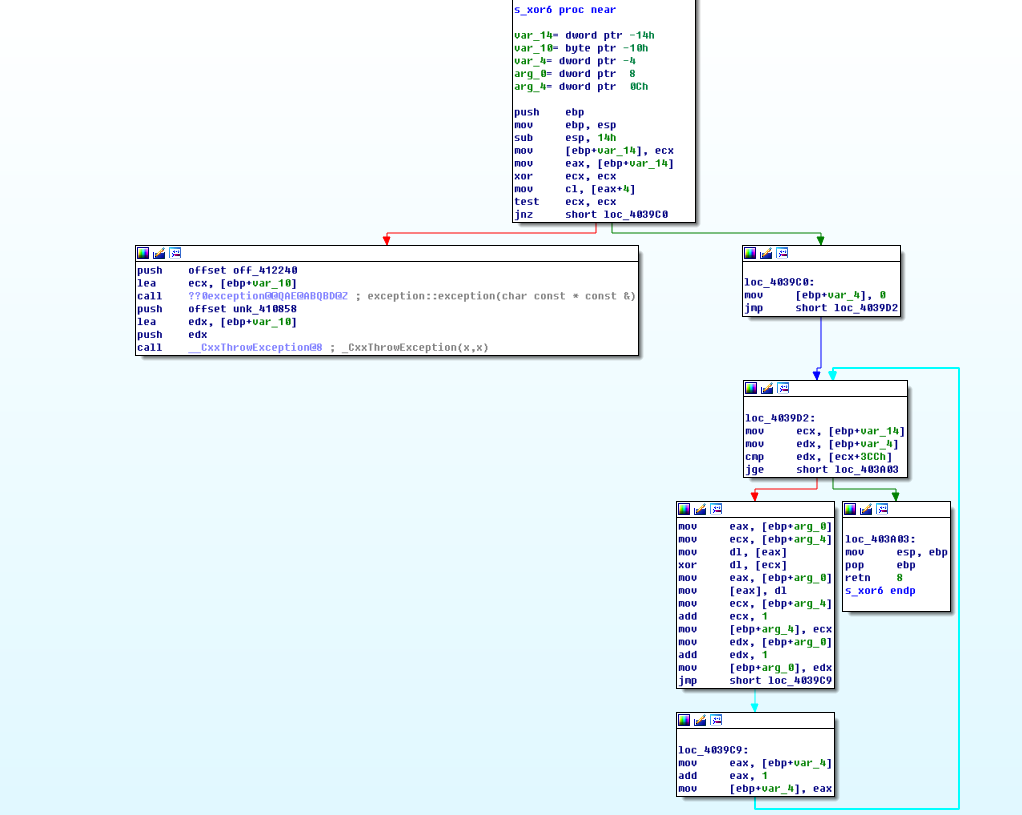
PEiD KANAL插件在一个相似位置显示了这个AES常量。PEiD识别的S和S-inv参考了S-box结构，它是一些加密算法的基本结构。

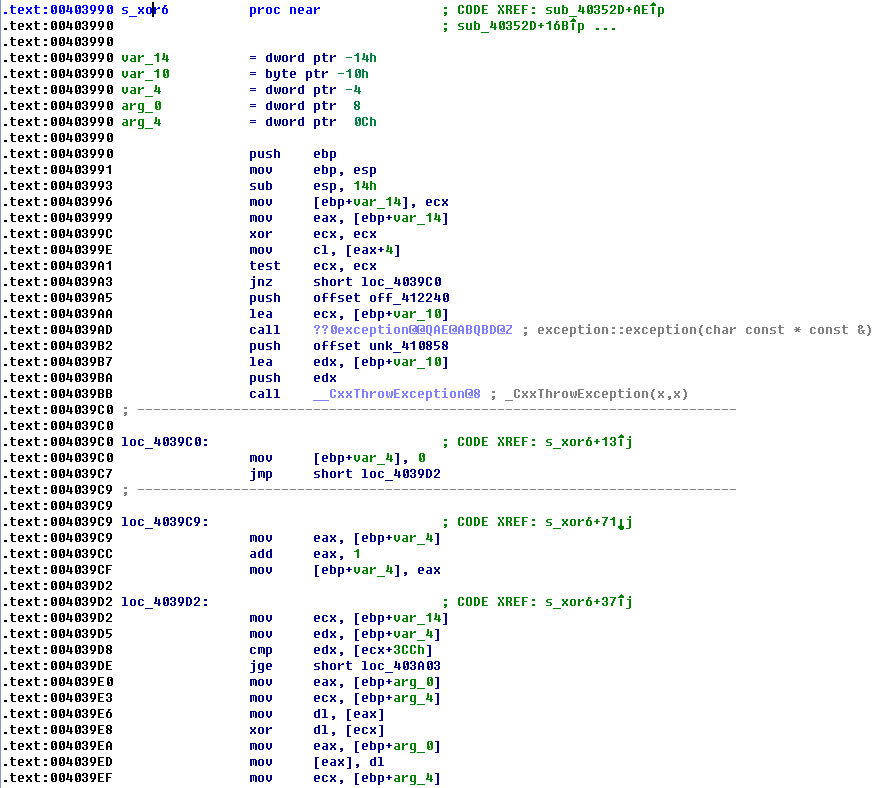


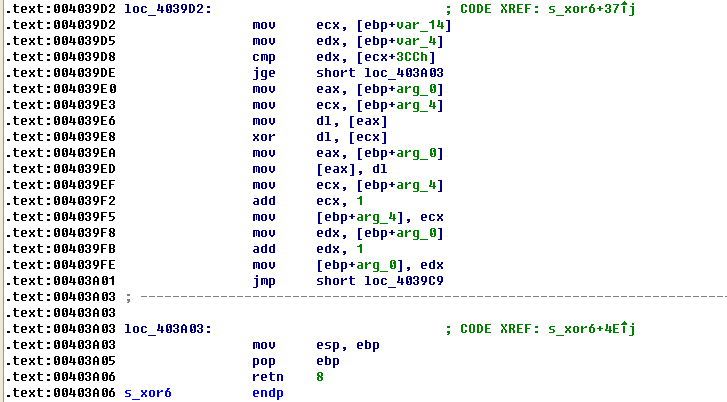
IDA熵插件显示了高熵的区域。首先，检查高8比特位的熵区域(256位的块大小的最小熵值是7.9)高亮显示了0x0040C900到0x0040CB00之间的区域——与前面识别的S-box区域相同。看一下高6比特位的熵区域(64位块大小的最小熵值为5.95),看到了.data段中从0x004120A3到0x004120A7之间一个区域。看到以0x004120A4开始的一个字符串包含所有Base64编码的字符：CDEFGHIJKLMNOPQRSTUVWXYZABcdefghijklmnopqrstuvwxyzab0123456789+/。

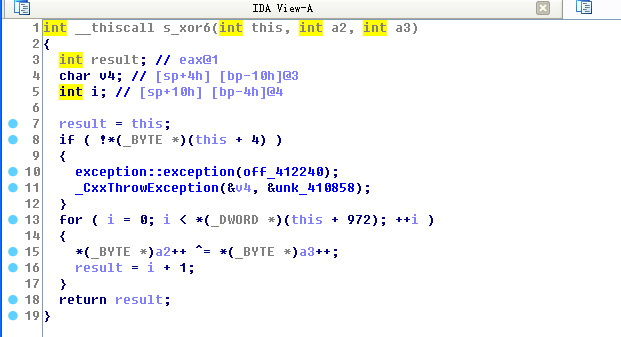
这不是标准的Base64编码字符串，因为大写字母AB和小写字母ab已经被转移到后面的大写或者小写段的后面。这个恶意代码使用了自定义的Base64加密算法。

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。



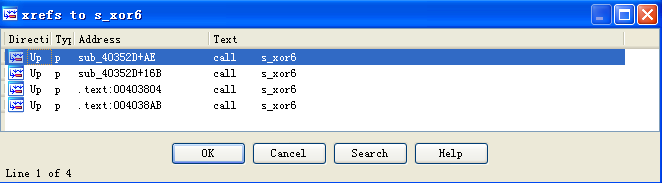






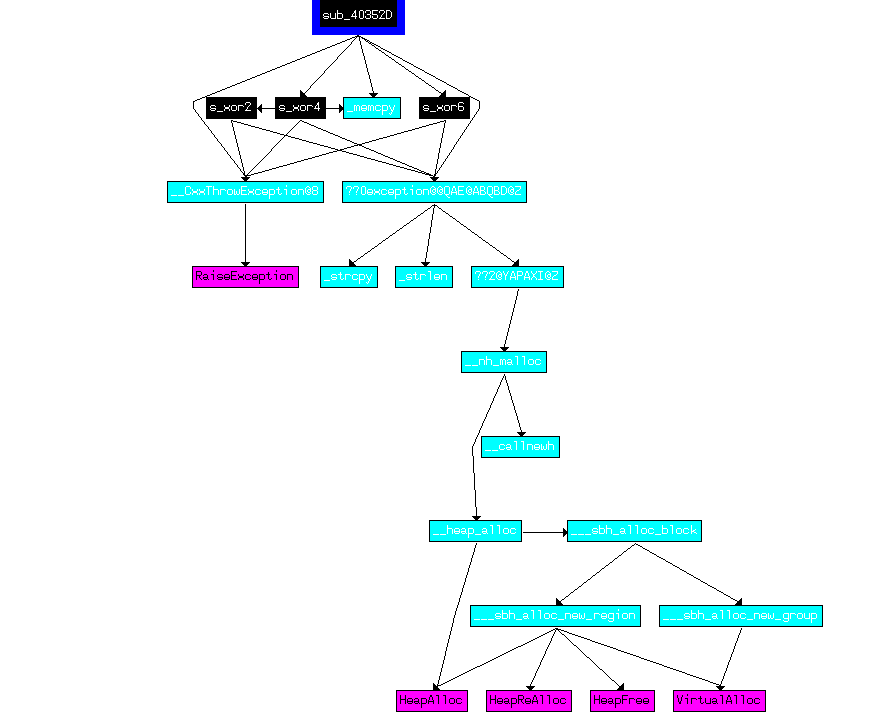
这里有一个循环，0x004039e8处的xor指令表明该函数是用于进行XOR加密的。arg\_0是一个指针，指向用来进行转换的原缓冲区，arg\_4也是指针，指向用来提供异或原数据的缓冲区。在循环中则是通过0x004039F2，0x004039f8指令对指向两个缓冲区的指针自增,通过004039cc对计数器变量var\_4自增。

查看s\_xor6的交叉引用



调用s\_xor6的函数的起始地址为0x0040352D。

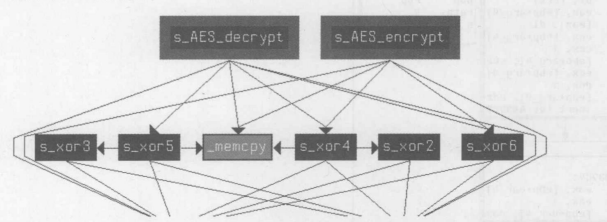
从0x0040352D开始，函数的交叉引用图如下



从这个图中，可以看出s\_xor6确实与其他AES加密的函数s\_xor2和s\_xor4相关。

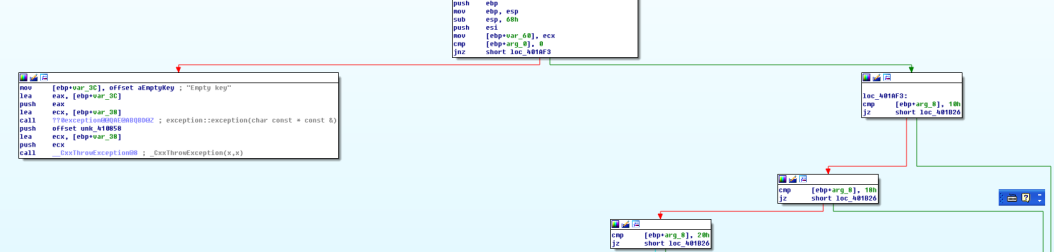
虽然有了s\_xor3和s\_xor5与AES解密相关的证据，但是这两个函数与其他函数的关系并不清晰。例如，当查找s\_xor5的交叉引用时，看到s\_xor5在两个位置(0x004037EE和0x0040392D)被调用，这个两个位置似乎包含有效的代码，但是这个区域并没有被定义为一个函数。这暗示当AES代码链接到恶意代码时，没有使用解密，因此解密例程最初显示为无用代码。

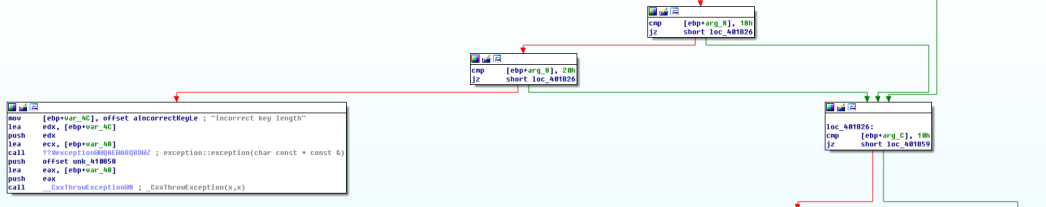
已经识别s\_xor5的函数(0x00403745)为解密函数，重新创建一个图，它显示从0x00403745(重命名为s\_AES\_decrypt)和0x0040352D(重命名为s\_AES\_encrypt)开始调用的所有函数。

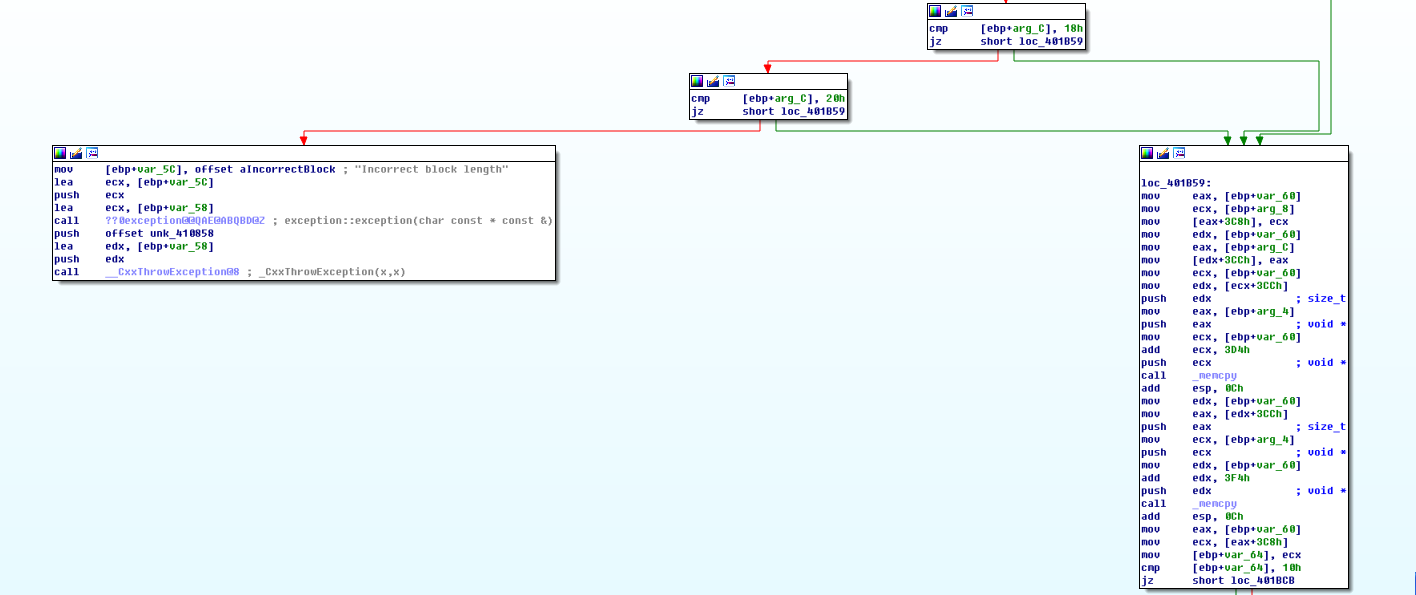


看到除了s\_xor1以外，所有XOR函数都与AES实现相关。

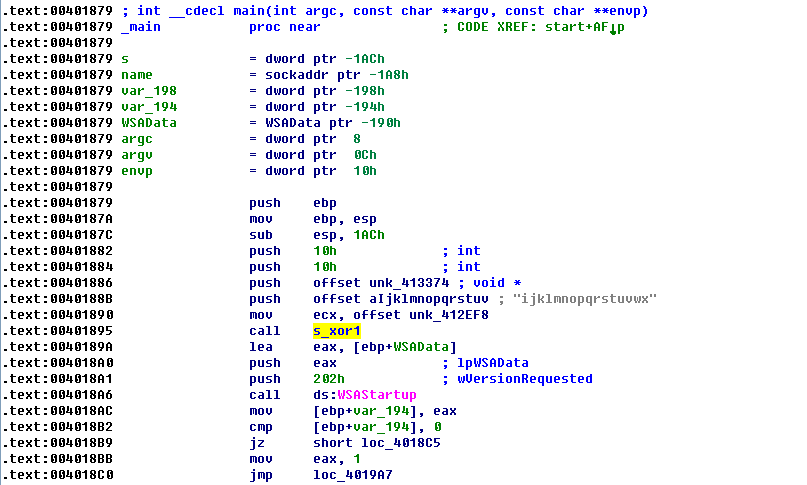
查看s\_xor1，看到当参数不正确时，开始出现几个分支，幸运的是恶意代码有错误信息显示。这些错误信息包括：空密钥，不正确的密钥长度以及不正确的块长度，这暗示了这是密钥初始化代码。



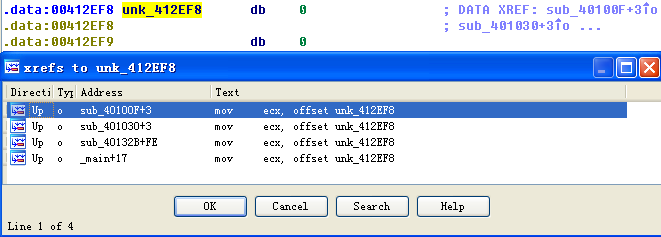




为了确认已经找到密钥的初始化代码，尝试找到这个函数与前面识别的AES函数之间的联系。查看s\_xor1的调用函数，看到恰好在s\_xor1被调用之前，存在unk\_412EF8的一个引用。用ECX将这个偏移量传入到s\_xor1函数。

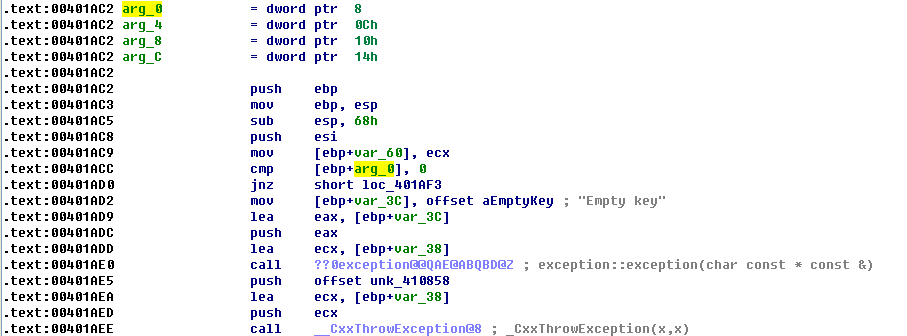


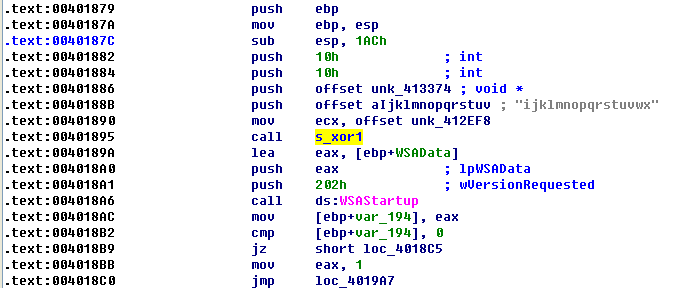
查看unk\_412EF8函数的其他引用，0x00401429（第三个）是unk\_412EF8的偏移量载入ECX的地方之一，它恰好在s\_AES\_encrypt调用之前被载入到ECX。地址unk\_412EF8必须是一个表示AES加密器的C++对象，并且s\_xor1是加密器的初始化函数。





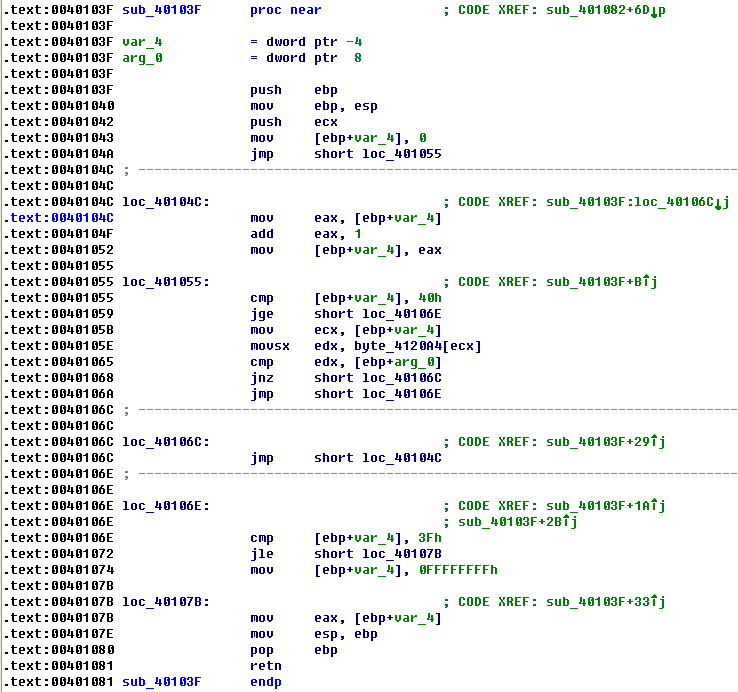
回到s\_xor1,看到在测试arg\_0参数以后，空密钥(Empty key)消息发出。可以推断arg\_0参数是一个密钥。看到这个参数在main中靠近s\_xor1的位置(0x401895)被设置，将arg\_0与被压入堆栈的字符串ijklmnopqrstuvwx关联在一起。在恶意代码中，这个字符串被用来AES加密。



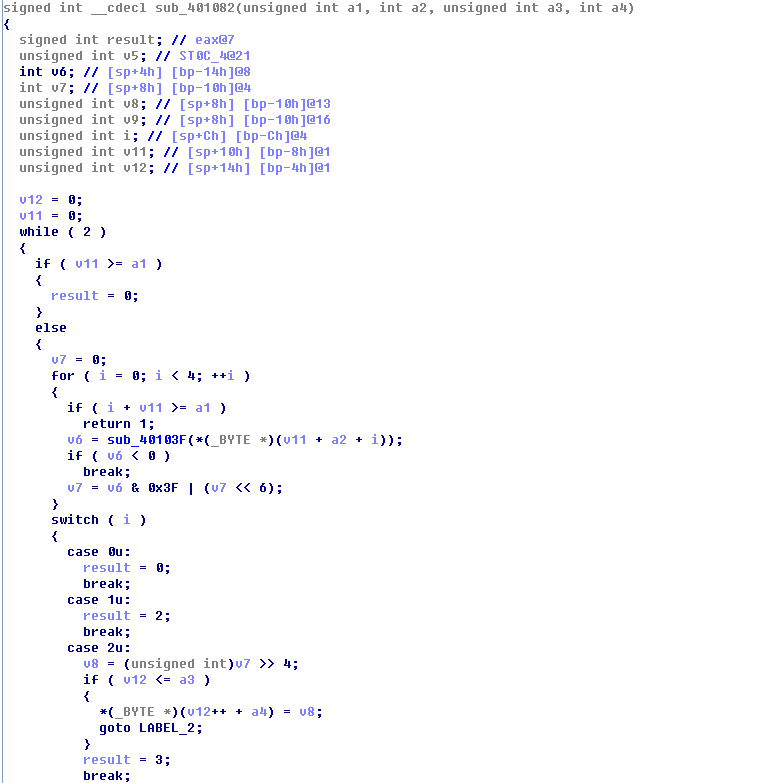


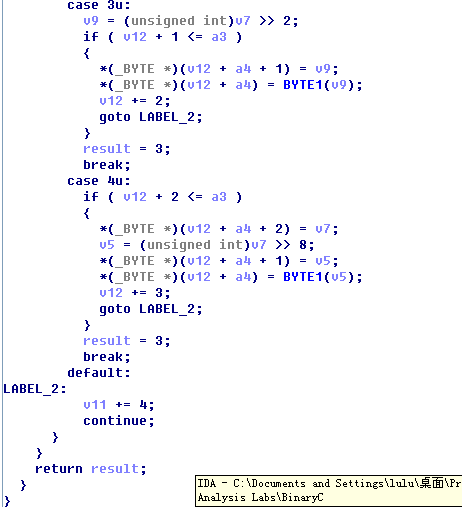
回顾一下AES是如何用于恶意代码中的：0x0040132B处的函数使用了s\_AES\_encrypt。加密在ReadFile与WriteFile调用之间发生；s\_xor1是AES初始化函数，它在进程启动时发生一次；s\_xor1设置的AES密码为ijklmnopqrstuvwx。

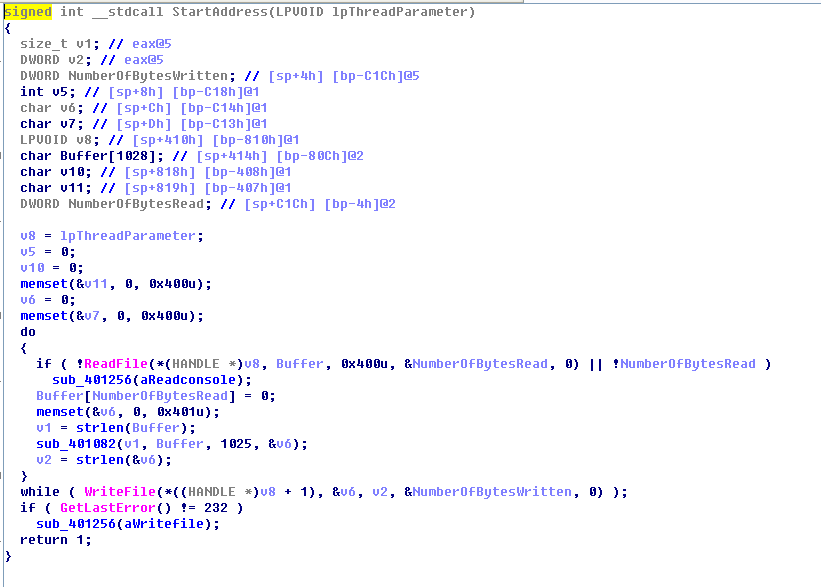
检查对字符串CDEFGHIJKLMNOPQRSTUVWXYZABcdefghijklmnopqrstuvwxyzab0123456789+/的引用，可知这个字符串在0x0040103F处的函数中。



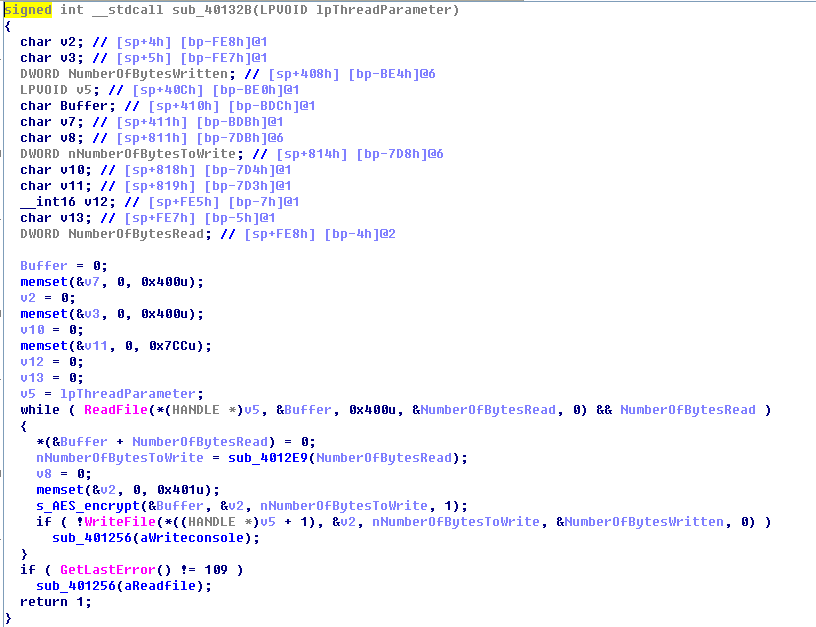
函数用索引查询这个字符串并且调用函数(在0x00401082处)将解密的字符串分成4字节的块。在0x00401082处的函数是一个自定义的Base64解码函数，并且可以看到位于ReadFile和WriteFile之间的解码函数(0x0040147C)调用了它。这与看到的AES用法具有相同模式，但是在不同的函数中。

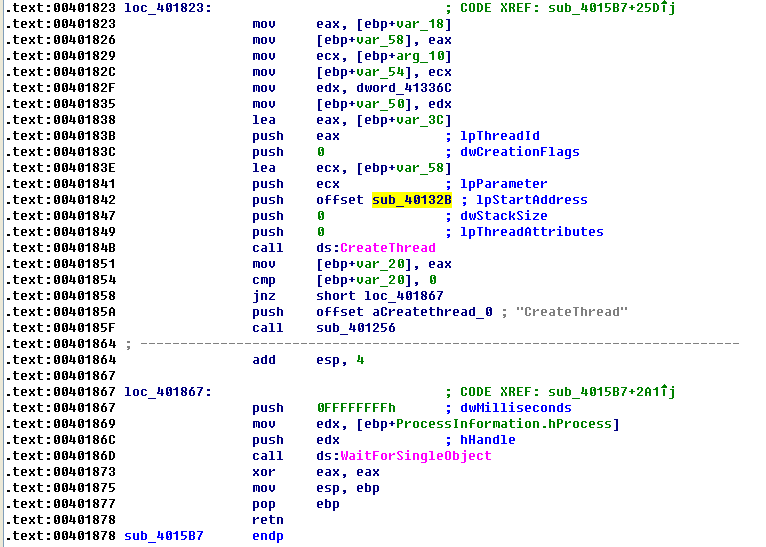






AES加密函数被0x0040132B开始的函数使用。看一下调用0x0040132B处函数的函数，看到0x0040132B是一个新线程的开始，这个新线程使用CreateThread创建，将0x0040132B重命名为aes\_thread。





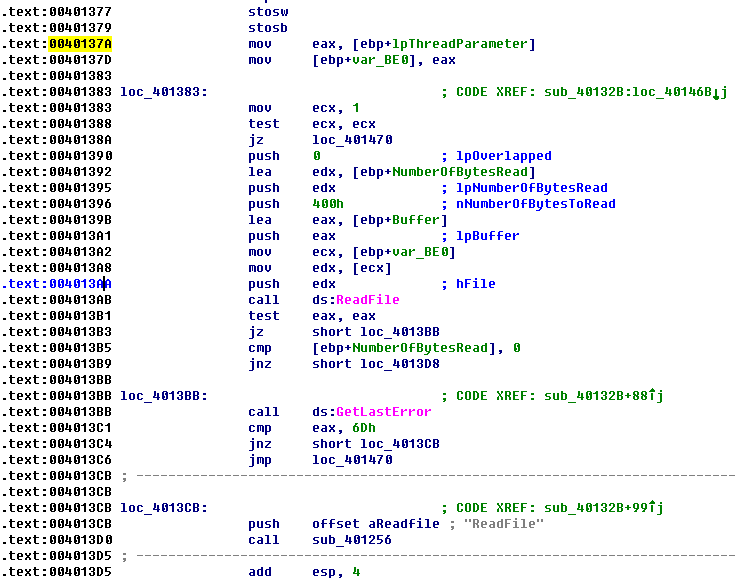
传递给线程开始函数的参数是var\_58的地址，并且看到有三个变量被压入到堆栈，它们与var\_58相对位置如下：

0x00401826处var\_18移入到var\_58中。

0x0040182C处arg\_10移入到var\_54中。

0x004035处dword\_41336C移入到var\_50中。

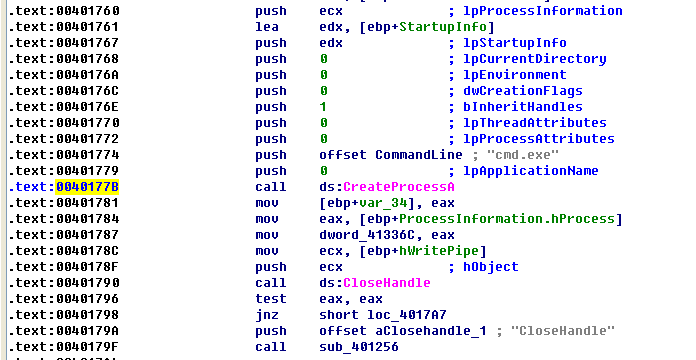
在aes\_thread(0x40132B)中，看到参数如何被使用。



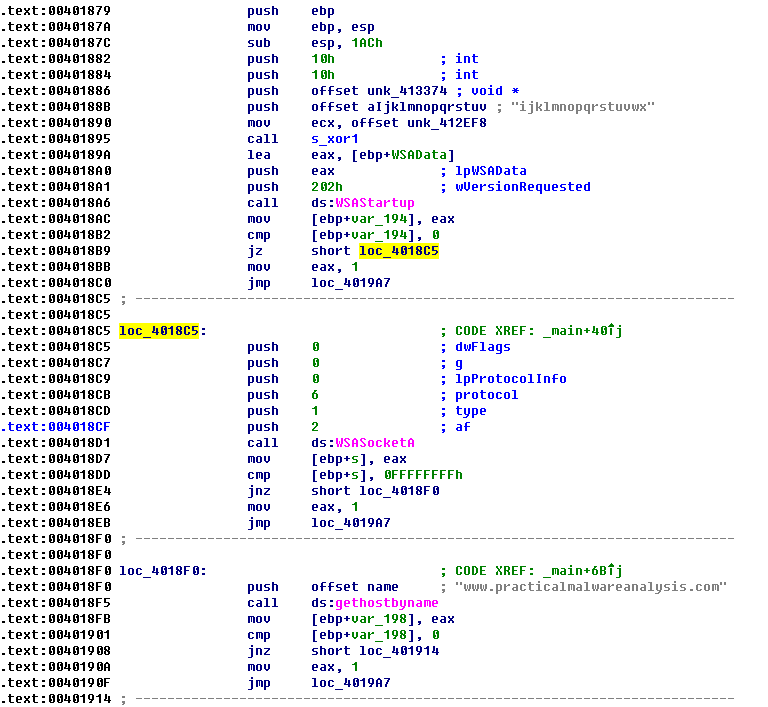


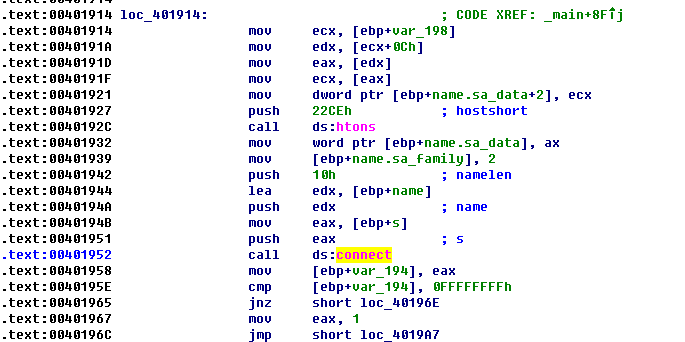
0x004013AA处的ReadFile压入堆栈的值向前映射到var\_58/var\_18，0x00401453处的WriteFile压入堆栈的值向前映射到var\_54/arg\_10。

跟踪句柄值，找到生成它的地方，发现var\_58和var\_18持有一个管道的句柄，早些时候，这个管道在函数0x0040132B中创建，并且它与一个shell命令的输出相连接。命令hSourceHandle被复制到shell命令的标准输出和标准错误，这条shell命令由0x0040177B处的CreateProcess命令启动。



在aes\_thread(var\_54/arg\_10)的WriteFile使用的另一个句柄可以被追溯到从\_main函数(0x00401879)中传递的参数——用connect调用创建的一个网络套接字。





函数aes\_thread (0x0040132B)读取shell命令的输出结果，在写入网络套接字之前加密它。

自定义的Base64加密函数(0x00401082)也在一个由它们宿主线程启动的函数(0x0040147C)中使用。利用一个假想的结论：Base64线程读取远程套机字的内容作为输入，经过函数解密后，它再将结果发送作为命令shell的输入。可以得出跟踪输入与跟踪AES线程的输入非常相似。

恶意代码中建立了两种类型的加密，尝试解密这些内容。首先解密自定义的Base64加密，假想从远处站点进行网络通信截获的部分信息是一个字符串：BInaEi==。

解密修改过的Base64编码的自定义脚本的实现。这是一个通用脚本，它可以被任何自定义的Base64编码实现。

|  |
| --- |
| Python import string  import base64  result = ""  ciphter\_content = "CDEFGHIJKLMNOPQRSTUVWXYZABabcdefghijklmnopqrstuvwxyz0123456789+/" standard\_b64 = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"  ciphter\_text = "BInaEi=="  for each\_ch in ciphter\_text:   if each\_ch in ciphter\_content:   result += starndard\_64[string.find(ciphter\_content, str(each\_ch))]   elif each\_ch == '=':   s += '='  result = base64.decodestring(result)  print(result) |

使用这个脚本，转换字符串来查看它发送给shell命令的命令是什么。攻击者发送一个罗列目录(dir)的请求。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。

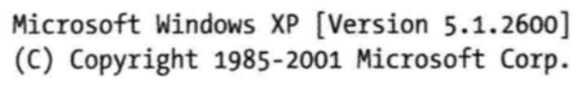
再尝试使用python解密AES的内容，在wireshark抓到的内容为



PyCrypto库提供了方便处理这样数据的加密例程。使用下面的代码，可以解密这些内容

|  |
| --- |
| Python from Crypto.Cipher import AES  import binascii  ciphter\_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"  ciphter\_text = binascii.unhexlify(ciphter\_text.replace(' ', ''))  obj = AES.new('ijklmnopqrstuvwx', AES.MODE\_CBC)  print(obj.decrypt(ciphter\_text)) |

raw变量包含了捕获到的原始网络内容；raw.replace删除原始字符串中的空格，并且函数binascii.unhexlify将十六进制的表示形式转换为二进制的字符串；的AES.new调用使用相应的密码和操作模式创建一个新的AES对象，供后面的解密调用使用。解密结果如下



加密陷阱

默认使用的PyCrypto库例程在Lab13-3中成功地工作，但是当试图直接实现解密例程时，存在一些潜在的漏洞，包含如下：

块加密算法拥有多种可能的操作模式，例如电子密码本(ECB)、密码段链接(CBC)以及加密反馈(CFB)等。每种模式要求在加密或者解密每个块时有不同的步骤，另外，一些模式还要求除了密码之外的一个初始化向量。如果你不符合这些实现条件，解密可能只部分工作或者根本不工作。

本实验中，直接提供了密钥。给定解密方案可能自己拥有生成假定用户提供的密钥或者基于字符串的密码的技术。在这种情况下，需要识别密钥生成算法并且分开备份。

在一个标准算法中，可能必须正确地指定选项。例如，一个单独的加密算法可能允许多个密钥大小、块大小、加密和解密轮数以及填充策略。

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，所以也没有什么东西暗示使用了加密。

1. **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?**

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。

1. **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?**

这三种技术都识别出了高级加密标准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。

1. **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。

1. **对于每一种加密技术，它们的密钥是什么?**

AES的密钥是ijklmnopqrstuvwx,自定义的Base64加密的索引字符串是：CDEFGHIJKLMNOPQRSTUVWXYZABcdefghijklmnopqrstuvwxyzab0123456789+/。

1. **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。

1. **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。

1. **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?**

查看详细分析过程。

1. **Yara**

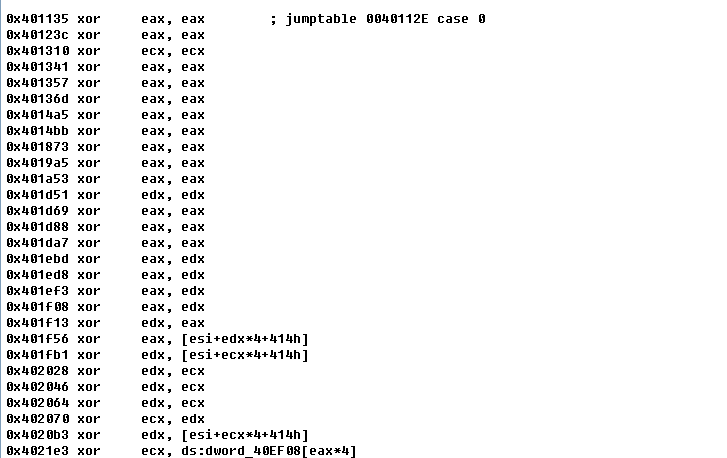
|  |
| --- |
| rule RuleforLab13\_01exe |
| { |
| meta: |
| description = "Lab13-01.exe" |
| strings: |
| $s1 = "Mozilla/4.0" fullword ascii |
| $s2 = "http://%s/%s/"fullword ascii |
|  |
| condition: |
| uint16(0) == 0x5a4d and |
| uint32(uint32(0x3c))==0x00004550 and filesize < 100KB and |
| all of them |
| } |
|  |
| rule RuleforLab13\_02exe |
| { |
| meta: |
| description = "Lab13-02.exe" |
| strings: |
| $s1 = "temp" fullword ascii |
| $s2 = "GDI32.dll" fullword ascii |
| $s3 = "GetLastActivePopup" fullword ascii |
| $s4 = "GetActiveWindow" fullword ascii |
| condition: |
| uint16(0) == 0x5a4d and |
| uint32(uint32(0x3c))==0x00004550 and filesize < 100KB and |
| all of them |
| } |
|  |
| rule RuleforLab13\_03exe |
| { |
| meta: |
| description = "Lab13-03.exe" |
| strings: |
| $s1 = "cmd.exe" fullword ascii |
| $s2 = "Empty key" fullword ascii |
| $s3 = "Incorrect key length" fullword ascii |
| $s4 = "Incorrect block length" fullword ascii |
| condition: |
| uint16(0) == 0x5a4d and |
| uint32(uint32(0x3c))==0x00004550 and filesize < 100KB and |
| all of them |
| } |



1. **IDA python**

查找使用xor指令的汇编代码

|  |
| --- |
| import idc |
| import idautils |
| for func in idautils.Functions(): |
| flags=idc.GetFunctionFlags(func) |
| if flags & FUNC\_LIB or flags & FUNC\_THUNK: |
| continue |
| dism\_addr = list(idautils.FuncItems(func)) |
| for line in dism\_addr: |
| m=idc.GetMnem(line) |
| if m=='xor': |
| op=idc.GetOpType(line,0) |
| if op==o\_reg: |
| print '0x%x %s' % (line,idc.GetDisasm(line)) |



1. **实验结论及心得体会**

通过完成Lab13-1、Lab13-2和Lab13-3，我深入学习了恶意代码分析的理论和实践。这一系列实验使我在静态和动态分析方面取得了重要的进展，加深了对恶意代码行为和加密技术的理解。

1. 静态分析的重要性：

在Lab13-1中，我通过使用静态分析工具（PEiD、IDAPython）对PE文件进行分析，识别了文件的基本属性、导入表、字符串和可能的加密手段。静态分析为我提供了恶意代码结构的初步认识，为后续的深入分析提供了基础。

1. 动态分析的关键作用：

Lab13-2的实验中，我深入学习了动态分析的过程，通过监控恶意代码的运行轨迹、文件和注册表的操作，以及与 C2 服务器的网络通信，揭示了攻击者的控制手段和可能的攻击目的。动态分析强调实际运行中的行为，对于发现隐藏的攻击行为至关重要。

1. 加密机制的深度剖析：

在Lab13-3的实验中，我学习了加密算法的静态识别方法，通过分析 xor 指令、高熵区域和自定义的 Base64 加密算法，揭示了恶意代码中的加密手段。这使我对加密的原理和在恶意代码中的应用有了更深层次的理解。

1. 实验的综合效果：

通过整合这三个实验，我不仅学到了分析恶意代码的基本原理和技术，还锻炼了自己的问题解决能力。从静态分析的PE文件基本信息到动态分析的行为监控，再到加密机制的深度分析，这一系列实验形成了一个完整的恶意代码分析流程。这种综合性的学习让我更好地理解了安全分析的复杂性和深度，为我今后在安全领域的深入研究打下了坚实的基础。