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

实验五：识别汇编中的 C 代码结构

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. 自学课本第六章内容。
2. 本章实验的目标是通过分析代码结构来理解一个程序的总体功能。每一个实验将指导发现与分析一个新的代码结构。每一个实验在前一个基础上构建，因此通过四种结构创建了一个复杂的恶意代码片段。
3. 完成了这个实验所需的工作后，应该能够当在恶意代码中遇到它们时，更容易地识别这些单独的结构。
4. 完成 Lab6 的实验内容，对 Lab6 的四个样本进行依次分析，编写 Yara 规则，并尝试使用 IDAPython 的自动化分析辅助完成。

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。

2.1.1 特点及优势

1. **深入分析**：IDA 允许研究人员深入地分析恶意软件的内部工作原理，识别其行为和功能。
2. **多平台支持**：IDA 支持多种处理器架构，如 x86、ARM、MIPS 等。
3. **图形界面**：IDA 提供了一个图形界面，使得代码的流程和结构更加直观，有助于理解恶意代码的执行流程。
4. **高级分析**：IDA 可以识别函数、局部和全局变量、类和其他高级结构。加速了分析过程。
5. **交互式**：用户可以在 IDA 中手动更改、注释和重命名变量和函数，以帮助理解代码。
6. **插件支持**：IDA 支持插件，允许用户扩展其功能。**其中 IDAPython 就是一个十分有用的插件，也是本实验主要使用的插件工具**。

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。

2.2.1 特点及优势

1. **自动化**：使用 Python 脚本可以自动化许多繁琐的任务，如标记特定的 API 调用、搜索特定的模式等。
2. **扩展性**：用户可以使用 Python 库来扩展 IDA 的功能，如进行数据分析、图形生成等。

3. **交互性**：IDAPython 允许用户在 IDA 的环境中交互式地运行 Python 代码，这对于快速测试和原型设计非常有用。

4. **快速响应**：当面对新的恶意软件样本时，分析师可以快速地使用 IDAPython 脚本来分析其行为，从而更快地响应威胁。

5. **定制化分析**：由于 IDAPython 的灵活性，分析师可以根据需要定制分析过程，以适应特定的恶意软件家族或攻击技术。

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	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 Lab06-01

3.2.1 静态分析

首先对可执行文件进行静态分析，首先使用 PEiD 查看其基本信息：

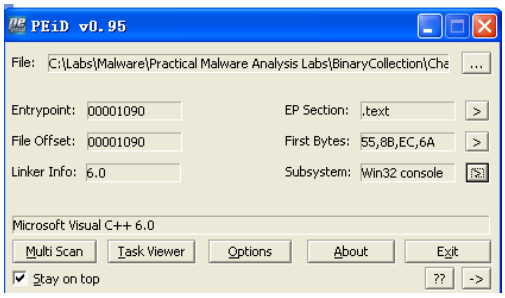


图 3.1: 基本静态分析

图3.1可以看到病毒并没有进行任何加壳处理，同样地是使用 VC6++ 进行的编写，接下来进一步查看其导入表：图3.2可以看到病毒有 WININET.DLL 以及其中的 InternetGetConnetedState 函数。这个 API 函数可以让应用程序通过 HTTP 协议访问 Internet 资源。**这代表着其可能有着查看本地系统 Internet 连接状态的作用**。

接着通过 PEView 查看其字符串：图3.3可以看到字符串 Error 1.1:NoInternet” 以及 “Success: Internet3Connection”。这暗示着**病毒可能会查看系统中是否存在可以利用的 Internet 连接便于其恶意行为，并输出相关字符串**。

3.2.2 实验问题

接下来，**将重点使用动态分析方法**。首先双击运行：

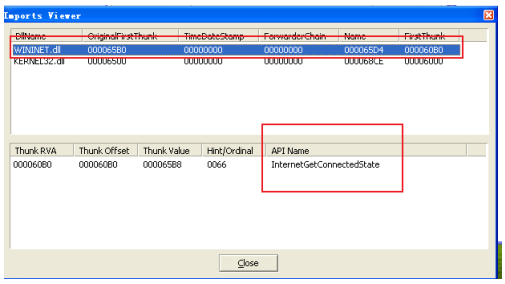
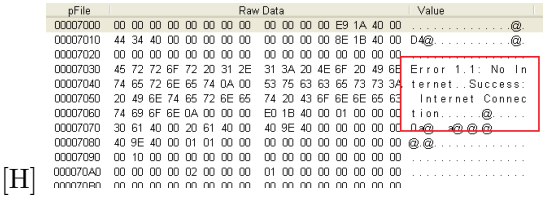


图 3.2: Lab06-01 导入表查看



[H]

图 3.3: Lab06-01 字符串

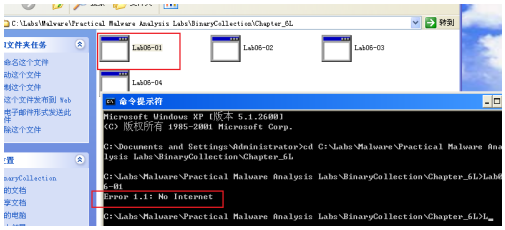


图 3.4: Lab06-01 运行结果

图3.4可以看到程序打印了一条“Success: Internet3Connection”便退出。因此还需要使用工具进行更深层次的动态分析。

1. **Q1: 由 main 函数调用的唯一子过程中发现的主要代码结构是什么?**

   回答： 是用 IDAPro 对文件进行加载分析，首先查看其 main 函数部分的调用流程图：

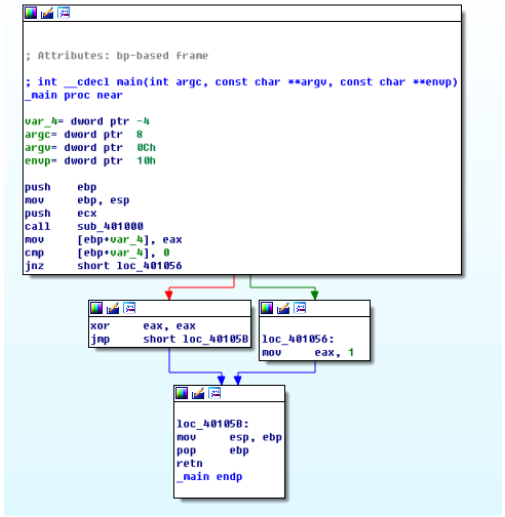


图 3.5: main 函数流程图

图3.4可以看到 main 函数部分的代码通过 call sub_401000 命令调用子过程 sub_401000，并且只有 mian 函数对其进行了调用。进一步查看 sub_401000 流程图：

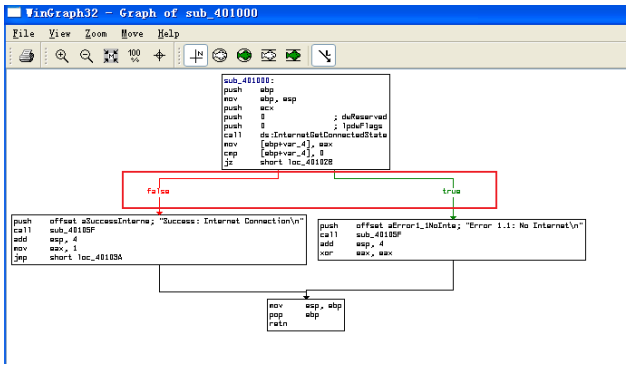


图 3.6: sub_401000 流程图

图3.6可以看到该函数通过 **call ds:InternetGetConnetcedState** 调用函数 InternetGetConnetcedState，根据函数的返回结果 eax 与 0 比较，进而通过 jz 命令进行控制流的分支。这是一个典型的 if 结构，因此**由 main 函数调用的唯一子过程主要代码结构是位于 0x401000 处的 if 语句**。

2. **Q2: 位于 0x40105F 的子过程是什么?**

   回答：从上面的流程图可以看到位于 0x40105F 的子过程，现在进一步查看其内容

```
.text:0040105F sub_40105F      proc near               ; CODE XREF: sub_401000+1C↑p
.text:0040105F                                         ; sub_401000+30↑p
.text:0040105F
.text:0040105F arg_0           = dword ptr  4
.text:0040105F arg_4           = dword ptr  8
.text:0040105F
.text:0040105F                 push    ebx
.text:00401060                 push    esi
.text:00401061                 mov     esi, offset stru_407098
.text:00401066                 push    edi
.text:00401067                 push    esi
.text:00401068                 call    __stbuf
.text:0040106D                 mov     edi, eax
.text:0040106F                 lea     eax, [esp+10h+arg_4]
.text:00401073                 push    eax             ; int
.text:00401074                 push    [esp+14h+arg_0] ; int
.text:00401078                 push    esi             ; FILE *
.text:00401079                 call    sub_401282
.text:0040107E                 push    esi
.text:0040107F                 push    edi
.text:00401080                 mov     ebx, eax
.text:00401082                 call    __ftbuf
.text:00401087                 add     esp, 18h
.text:0040108A                 mov     eax, ebx
.text:0040108C                 pop     edi
.text:0040108D                 pop     esi
.text:0040108E                 pop     ebx
.text:0040108F                 retn
.text:0040108F sub_40105F      endp
.text:0040108F
```

图 3.7: 0x40105F

图3.7处的代码并不能给我给我们直接的提示，因此我们回到图3.6中。可以看到在调用 0x40105F 前，有一个格式化的字符串被压入栈中，并且结尾还有一个换行符"\n"。这都可以让我们推断出**位于 0x40105F 处的子例程是 printf**。

3. **Q3：这个程序的目的是什么?**

   回答：为了能更好地推断出恶意代码的目的，将其重命名为 printf:

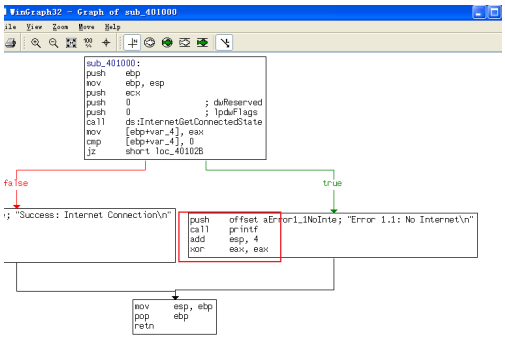


图 3.8: Lab06-01 目的分析

图3.8可以看到将 printf 修改命名后，函数到的逻辑就十分清晰了。最后 printf 被调用后，eax 的值被设为 0 或者 1，函数返回。**故综上所述，该程序目的是在连接 Internet 之前，首先检查是否存在一个可用的 Internet 连接。如果找到可用连接，就通过 printf 输出打印 “Success: Internet3Connection”；否则打印 “Error 1.1:NoInternet”**。

3.3 Lab06-02

3.3.1 静态分析

首先使用 PEView 观察其字符串：

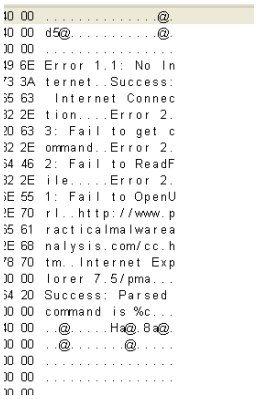


图 3.9: 观察字符串

图3.9可以看到有许多需很有指示性的字符串。其中有着著名的网址彩蛋 www.practicalmalwareanalysis.com 作为网络特征，还有三条很起眼的报错字符串：

- Error 2.3: Fail to get command
- Error 2.2: Fail to ReadFile
- Error 2.1: Fail to OpenUrl

由此可以推断出该程序很可能使用 Internet Explorer 作为浏览器，打开 http://www.practicalmalwareanalysis 网址，若成功，显示 Parsed command 解析指令。接下来使用 PEiD：

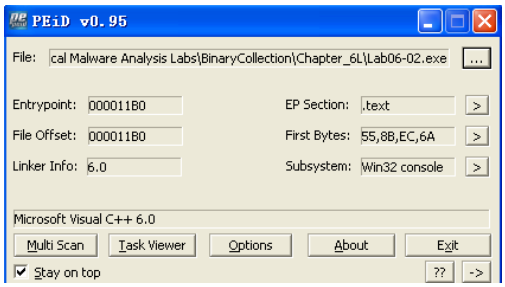


图 3.10: PEiD 查看

图3.10可以看到该病毒也没有进行加壳，使用 VC6++ 编写，接下来查看导入表：

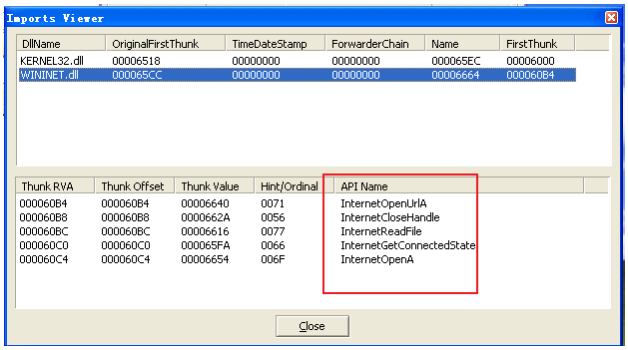


图 3.11: 查看导入表

图3.11可以看到一些很具有指示性的导入函数。首先再次出现的 WININET.dll 暗示存在网络行为，然后重点分析几个函数：

- InternetOpenA: 用于初始化一个用于 Internet 访问的会话句柄。可能使用此函数来建立与互联网的连接，以进行后续的网络活动，如下载文件、传输数据等。
- InternetReadFile: 用于从指定的 URL 下载文件的函数。病毒可能下载恶意文件或更新自身的组件。
- InternetOpenUrlA: 用于打开指定 URL 的连接，并返回一个可以用于读取 URL 内容的句柄。病毒可能使用此函数来访问远程服务器，以获取指令、下载恶意文件或上传被感染的文件。
- InternetCloseHandle: 用于关闭由 InternetOpenUrlA 和 InternetOpenA 打开的句柄。此函数可能被用来清理其使用的资源，以避免被检测到或留下痕迹。

可以看到真的是有始有终，恶意代码使用了这一整套 API 函数，显示初始化了对 WinNet 的使用，还设置了 HTTP 的 User-Agent 字段，最后甚至也会关闭句柄释放资源。**可以说是滴水不漏，佩服佩服**。

3.3.2 实验问题

接下来，**将重点使用动态分析工具**。

首先对其网络行为进行捕捉，使用 ApateDNS 重定向工具定向到本机 localhost 即 127.0.0.1。然后使用 Netcat 监听其 HTTP 的 80 的端口，双击运行后发现：

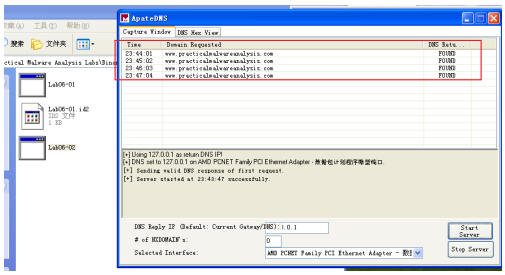


图 3.12: ApateDNS 重定向捕捉

图3.12可以看到通过重定向，**能够成功捕捉到多条来自 www.practicalmalwareanalysis.com 的 DNS 请求。进而使用 URL 进行请求网页**。但是使用 Netcat 没有监听到任何异常，推测可能是网络上的波动等原因。因此接下来重点使用 IDA Pro 进行深度分析：

1. **Q1:main 函数调用的第一个子过程执行了什么操作?**

   回答：首先使用 FlowChart 流程图直接查看 main 函数的调用流程：

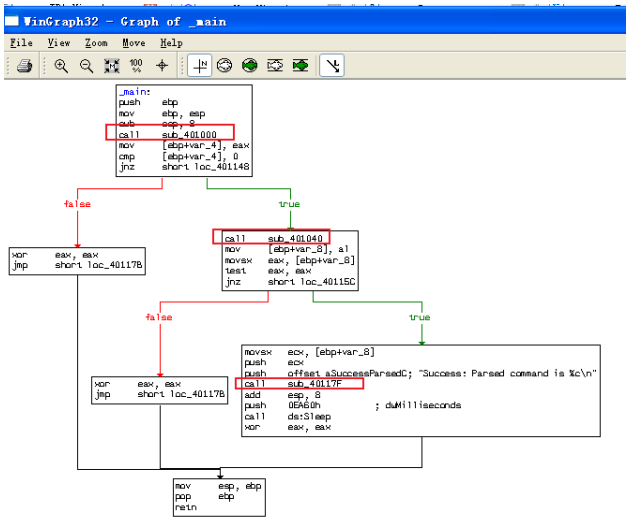


图 3.13: Lab06-02 的 main 函数调用流程

图3.13可以看到 main 函数调用了三个值得注意的地方：

- 0x401000：其中这个部分和 Lab06-01 相同，并且是直接调用的第一子进程；
- 0x401040: 新出现；
- 0x40117F: 新出现，间接调用；

题目问的是第一个子过程的操作，因此来到第一个子过程 sub_0x401000 处。

```
text:00401000 ; ============== S U B R O U T I N E =======================================
text:00401000
text:00401000 ; Attributes: bp-based frame
text:00401000
text:00401000 sub_401000      proc near               ; CODE XREF: _main+6↓p
text:00401000
text:00401000 var_4           = dword ptr -4
text:00401000
text:00401000                 push    ebp
text:00401001                 mov     ebp, esp
text:00401003                 push    ecx
text:00401004                 push    0               ; dwReserved
text:00401006                 push    0               ; lpdwFlags
text:00401008                 call    ds:InternetGetConnectedState
text:0040100E                 mov     [ebp+var_4], eax
text:00401011                 cmp     [ebp+var_4], 0
text:00401015                 jz      short loc_40102B
text:00401017                 push    offset aSuccessInterne ; "Success: Internet connection\n"
text:0040101C                 call    sub_40117F
text:00401021                 add     esp, 4
text:00401024                 mov     eax, 1
text:00401029                 jmp     short loc_40103A
text:0040102B ; ---------------------------------------------------------------------------
```

图 3.14: 子进程 sub_0x401000

图3.14可以看到此时的代码和 Lab06-01 相同。故**执行的操作是类似 if 语句的形式，判断是否有可能使用的 Internet 连接，然后进行分支跳转**。

2. **Q2: 位于 0x40117F 的子过程是什么?**

   回答：这里如果直接查看其反汇编可能也找不到一些有用的信息，因此首先来到调用它之前的位置：

```
text:0040115C ; ---------------------------------------------------------------------------
text:0040115C
text:0040115C loc_40115C:                             ; CODE XREF: _main+26↑j
text:0040115C                 movsx   ecx, [ebp+var_8]
text:00401160                 push    ecx
text:00401161                 push    offset aSuccessParsedC ; "Success: Parsed command is %c\n"
text:00401166                 call    sub_40117F
text:0040116B                 add     esp, 8
text:0040116E                 push    0EA60h          ; dwMilliseconds
text:00401173                 call    ds:Sleep
text:00401179                 xor     eax, eax
text:0040117B
text:0040117B loc_40117B:                             ; CODE XREF: _main+16↑j
text:0040117B                                         ; _main+28↑j
text:0040117B                 mov     esp, ebp
text:0040117D                 pop     ebp
text:0040117E                 retn
text:0040117E _main           endp
text:0040117E
```

图 3.15: 0x40117F

图3.15可以看到有一个参数 “Success: Parsed command is %c\n” 入栈，这样明显的字符串” \n “提示我们这是一个完整的句子，而**%c 又是典型的格式化字符，会被替换为另一个被压入栈的参数。这提示我们这是格式化字符串。而调用以格式化字符串为参数的，自然最常见的就是 printf 了，因此位于 0x40117F 的子例程也是 printf**。

3. **Q3: 被 main 函数调用的第二个子过程做了什么?**

   回答：由上面的 main 函数流程图可知，另一个被调用的子过程是 0x401040，查看其反汇编，这里有所有导入表的函数。都进行了调用：

```
.text:00401040 hInternet      = dword ptr -0Ch
.text:00401040 dwNumberOfBytesRead= dword ptr -8
.text:00401040 var_4          = dword ptr -4
.text:00401040
.text:00401040                push    ebp
.text:00401041                mov     ebp, esp
.text:00401043                sub     esp, 210h
.text:00401049                push    0               ; dwFlags
.text:0040104B                push    0               ; lpszProxyBypass
.text:0040104D                push    0               ; lpszProxy
.text:0040104F                push    0               ; dwAccessType
.text:00401051                push    offset szAgent  ; "Internet Explorer 7.5/pma"
.text:00401056                call    ds:InternetOpenA
.text:0040105C                mov     [ebp+hInternet], eax
.text:0040105F                push    0               ; dwContext
.text:00401061                push    0               ; dwFlags
.text:00401063                push    0               ; dwHeadersLength
.text:00401065                push    0               ; lpszHeaders
.text:00401067                push    offset szUrl    ; "http://www.practicalmalwareanalysis.com"...
.text:0040106C                mov     eax, [ebp+hInternet]
.text:0040106F                push    eax             ; hInternet
.text:00401070                call    ds:InternetOpenUrlA
.text:00401076                mov     [ebp+hFile], eax
.text:00401079                cmp     [ebp+hFile], 0
.text:0040107D                jnz     short loc_40109D
.text:0040107F                push    offset aError2_1FailTo ; "Error 2.1: Fail to OpenUrl\n"
.text:00401084                call    printf
```

图 3.16: 0x401040 处反汇编 1

图3.16可以看到 0x401040 处：

- 首先将 Internet Explorer 7.5/pma 入栈,这是浏览器的使用 Agent,然后通过 **InternetOpenA** 初始化一个用于 Internet 访问的会话句柄和对 WinNet 库的使用。
- 接下来又是一个经典的网址 URL 入栈：www.practicalmalwareanalysis.com，然后使用 InternetOpenUrl 打开概静态网页，**ApateDNS 捕捉到的 DNS 请求与之有关**。
- 然后函数 InternetOpenUrl 返回的结果 eax 赋值给了 hFile，然后将 hFile 和 0 进行比较，**若 hFile 等于 0，则证明其调用失败。令对应失败的字符串入栈，然后调用刚才分析的 0x40117F 即 printf 在命令行输出**。

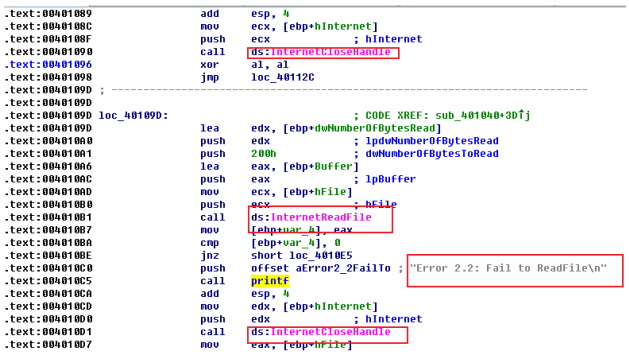


图 3.17: 0x401040 处反汇编 2

图3.17可以看到：

- 此时若之前的 printf 输出完错误信息后，这里会直接调用 InternetCloseHandle 关闭释放资源。
- 若之前的 hFile 不等于 0，则会使用 InternetReadFile 从 InternetOpenUrlA 打开的静态网页读内容。

现在重点分析 InternetReadFile 来获取其详细做了什么：读取内容时，可以发现其在调用该函数之前，又是几个参数入栈。前面的参数 dwNumberOfBytesToRoad 即 200h 字节是具体读取的长度。而后面的 **Buffer 参数**是一个保存数据的数组，类似于**计算机网络课上所学到的缓冲区的概念**。

因此推测该函数可能是会从下载的 HTML 读取内容，具体读取内容的长度为 0x200h 字节，读取后的数据存放在 Buffer 中。

接下来分析其函数 InternetReadFile 调用后的行为：可以看到其调用后将返回值 eax 先放入某个地址后与 0 对比：

- 若结果是 0，则和之前一样通过**压入错误字符串提示" Error 2.2... "，提示没有成功地读取文件内容，然后调用 printf 输出这个格式化字符串**，最后通过 InternetCloseHandle 关闭句柄释放资源。
- 若结果不是 0，则证明函数调用成功，继续执行，会通过 jnz 命令体哦转到地址 loc_4010E5 处。到该位置查看：

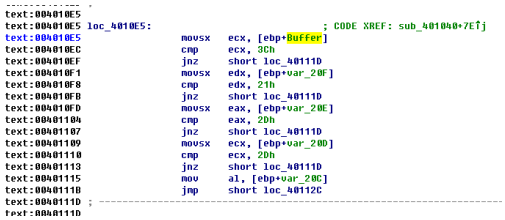


图 3.18: loc_4010E5 处

图3.18可以看到该位置不断地利用之前读取到网页内容存取的缓冲区数组 Buffer。然后这里将其逐步通过类似于数组名作为首地址的方式不断 +1，后裔位置，实际起到的作用就是**将缓冲区中读到的内容逐一地拿出，保存到一个寄存器中，依次是 edx，ecx 等，然后每次都与另一个字符进行比较**。

接下来具体分析其比较的内容，将每次比较的字符串如 0x3Ch 等换成实际的 ASCII 字符，然后可以看到：

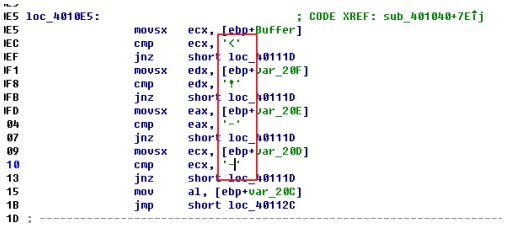


图 3.19: loc_4010E5 处对比字符内容

图3.19可以看到其正好拼成了 **<!–，这是标签语言 HTML 的注释开头部分**。因此总结下来，就是恶意代码会下载网页的内容即 Buffer 数组中的内容依次对比，通过四条 cmp 指令判断石佛指向的是某个 HTML 注释的开头，即检测是否是网页最开始位置的一条 HTML 注释。

若其中比较有一个位置不成功，跳转到 loc_40111D 处：

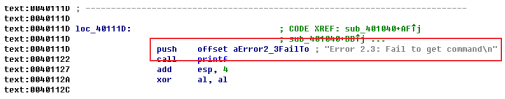


图 3.20: loc_40111D

图3.20可以看到此处位置又是同样地提示错误”Error2.3:Fail to get command”，然后 printf 输出返回。**但是如果前面的全部匹配，即网页由注释开头，则会跳转到 loc_40112C 处**。

在回答下一个问题之前，先对前面内容分析进行总结，因此该恶意代码调用的第二个子过程 0x401040，会使用我们之前静态分析发现的诸多网络 API 函数和错误提示字符串。结合使用一组 API 依次初始化，下载位于 http://www,practicalmalwareanalysiscom/cc.html 的网页，将网页内容暂存到缓冲区，最后判断该网页是否是以 HTML 注释开头。其中任何一个 API 函数没有成功执行都会提示错误信息，然后释放句柄资源。

4. **Q4: 在这个子过程中使用了什么类型的代码结构?**

   回答：

接下来继续分析。由于上面想要继续分析，下一条语句是 **mov al, [ebp+var_20C]，而通过上下文可知，这个实际上就是 Buffer。不过是匹配成功后的后续处理**

为了方便分析，最好的方式就是令 **IDA 能够正确地意识到 Buffer 实际上是作为 array 出现的，而不仅是一个叫做 var_2C 的局部变量。因此首先操作缓冲区：**

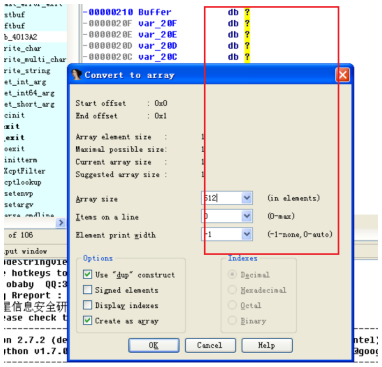


图 3.21: 操作 Buffer 缓冲区

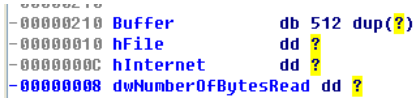


图 3.22: 定义 Buffer 为数组

图3.21和图3.22可以看到在 IDA 中定义缓冲区的大小，右键单击缓冲区的开始位置，选择 Define array，然后输入所需的大小和元素类型。定义为 512 字节的数组。此时回到刚才的 0x4010E5 处：

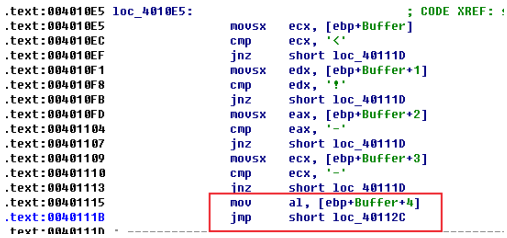


图 3.23: 正常显示 Buffer

图3.23可以看到能正常显示 [ebp+Buffer+4] 了。**故如果前面索引 0-3 位置能够匹配" <!– "，则后面会取出第五个放入 AL 中继续处理**

总的来说，该恶意代码调用的第二个子过程 0x401040 使用 InternetReadFile 读取数据并存如一个字符数组 Buffer，然后每次一个字节地对这个数组进行比较，以对比一个 HTML 注释。

5. **Q5: 在这个程序中有任何基于网络的指示吗?**

   回答：

   对之前的分析进行回顾，可以发现该恶意代码有两条网络特征：

   - 恶意代码使用 Internet Explorer 作为浏览器，和 7.5/pma 组成其 HTTP 的 User-Agent 字段;

- 使用一系列 API 函数从 http://www.practicalmalwareanalysis.com/cc.htm 下载网页并存如缓冲区。

6. **Q6: 这个恶意代码的目的是什么?**

   回答: 回到 main 函数继续分析:

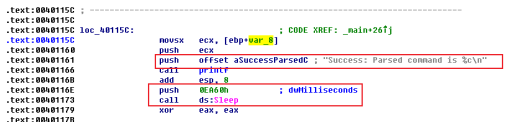


图 3.24: main 函数冬眠行为

图3.24可以看到, 基于之前的分析, 0x401040 函数返回后, 若返回值不为 0, 则证明正常返回。(若为 0, 同样地回去提示错误信息)

然后压入 ecx, 实际上是产生的一个 HTML 解析之后的字符, 再压入" Success: Parsed command is ", 将 ecx 填入其后。

然后可以看到函数会使用压入栈的 dwMilliseconds=0XEA60 即 60000 毫秒即一分钟, 调用 sleep 进行休眠。

对所有分析进行总结, 这个恶意代码首先导入了一系列 API 函数, 判断是否存在一个可用的 Internet 连接, 使用一个独特的来自 Internet Explorer 的 User Agent 下载一个网页。这过程任何一步错误就会输出对应的错误信息后终止运行。获取到网页数据后, 该网页包含了一段由 <!–开始的 HTML 注释, 通过缓冲区的数据进行一一对比来筛查, 最后再将 <!–后的第一个即第五个字符输出到屏幕输出格式是 "Success: Parsed command is" 后面。这一系列工作结束后, Lab06-02 会结束一天的工作, 开始呼呼大睡, 长达 1 分钟后, 离开人世。

3.4 Lab06-03

3.4.1 静态分析

同样先进行静态分析, 首先使用 PEView 查看其字符串:

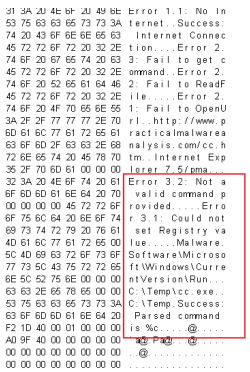


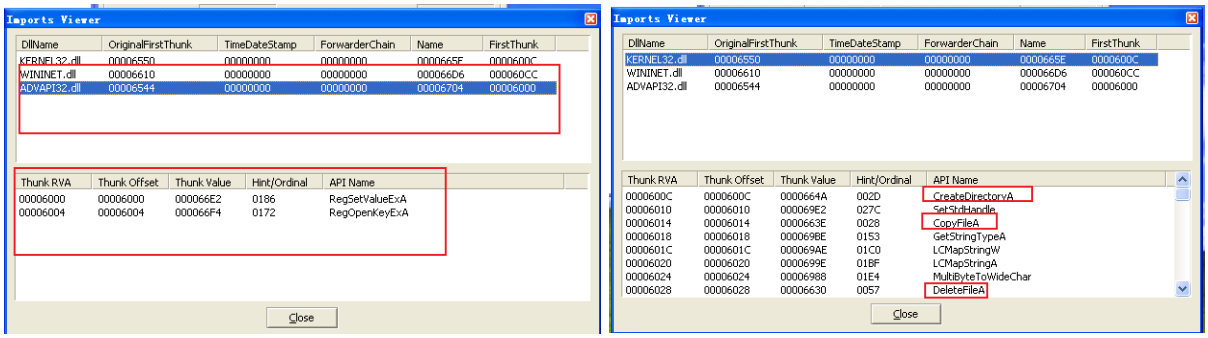
图 3.25: strings

图3.25可以看到此时除了 Lab06-02 出现的那些错误提示外, 新出现了其他字符串, 包括新的错误提示: Error3.1 和 Error3.2。其中重点可以发现: "Software\Microsoft\Windows\CurrentVersion\Run" 这是前面再 Lab3 中就出现过的注册表路径 **autorun**。具体而言:

- autorun 是一个用于指定在 Windows 系统启动时自动运行的程序或脚本的位置。在这些位置下，用户和系统级别的程序可以注册自己的启动项。
- 当 Windows 系统启动时，它会自动检查这些位置，并运行注册的程序或脚本。这样，用户或系统所需的程序可以在系统启动时自动加载和运行，提供所需的功能或服务。
- 一些恶意软件或病毒也会利用 Autorun 位置来注册自己，以在系统启动时自动运行并执行恶意行为。

因此**推测该恶意代码有修改注册表的行为**。然后 C:\Temp\cc.exe 是一个目录下的文件，**可以用作一个有效的本次特征，便于后续的 Yara 规则检测识别**。

然后使用 PEiD，发现其仍没有进行加壳，然后看起导入表：



导入表 1　　导入表 2

图 3.26: 查看导入表

图??可以看到新出现了一些导入表，其中 Delete,Copy 和 Create 用来删除、赋值和创建文件或者目录。而 **RegOpenKeyExA 和 RegSetValueExA 通常一起出现来向注册表插入信息，因此初步怀疑是向 autorun 位置插入实现开机自启动的持久化运行**。

3.4.2 实验问题

由于网络行为之前的 Lab06-02 使用 NETCAT 已经受挫了，这里直接使用 IDA 开干。进行动态分析。

1. **Q1: 比较在 main 函数与实验 6-2 的 main 函数的调用。从 main 中调用的新的函数是什么？**

   回答：直接使用 Main 函数的流程图可以发现，**在 0x401000 和 0x401040 处的函数与 Lab6-2 的一样。而正如之前分析的，0x401000 是用来检查是否存在一个有效的 Internet 连接的，而 0x401040 是通过下载网页然后解析 HTML 注释并返回的**。但有两个函数是 Lab06-02.exe 没有的：

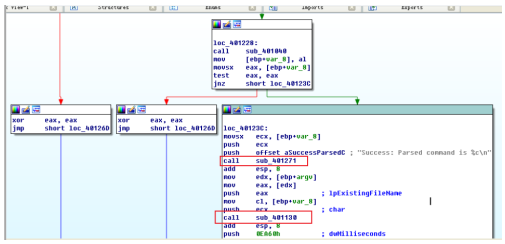


图 3.27: 新出现的 0x401130 和 0x401271

图3.27可以看到两个函数 **0x401130 和 0x401271 是 02 没有的**。进入两个函数调用前后反汇编：

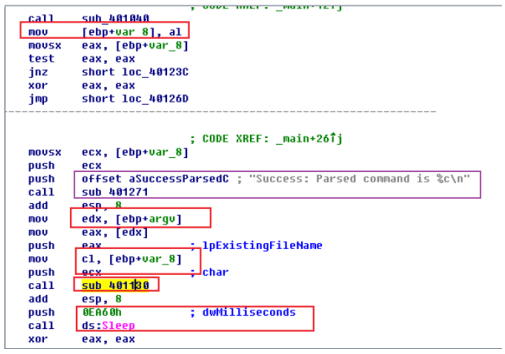


图 3.28: 0x401130 和 0x401271 调用前后

对于函数 0x401271，和之前分析 Lab06-01 和 Lab06-02 同样的道理，这里观察到在调用之前将字符变量 aSuccessParsedC 压入了栈中，而该字符变量实际上是 “Success: Parsed command is %c \n”。再次出现的”\n” 这个格式化字符提示我们 **0x401271 也是 printf**。将其改名。**因此，再次出现了 0x401000 和 0x401040 这两个 Lab06-02 中的函数，但是仍有两个不一样的。0x401271 处的是 printf 实际上也是之前出现过的函数，而 0x401130 位置的函数全新出现，从下一问开始探究**。

2. **Q2: 这个新的函数使用的参数是什么?**

   回答：从 Q1 可知，0x401000 和 0x401040 这两个之前就有，printf 也有。因此这里的新的函数应该唯一指的是 **0x401130 位置的函数**。

   图3.28可以看到调用前后传入了两个参数 argv 和 var_8：

   - argv：这参数在 Java 中经常出现，其实就是标准 main 函数的参数中的 argv[0]，即当前运行程序的名字，也就是对 **Lab06-03.exe 这个字符串的引用**。
   - var_8: 这个参数通过语句 mov [ebp+var_8], al 可以发现其就是由 al 赋值而来，而 al 是 eax 的低 8 位，eax 又是上一个函数即 0x401040 的返回结果，因此推测 var_8 包含的就是用于下载和解析 HTML 注释的函数返回结果，也就是**解析 HTML 出来的指令字符**。

3. **Q3: 这个函数包含的主要代码结构是什么?**

   回答：接下来来到函数位置查看 0x401130 位置其具体代码结构：

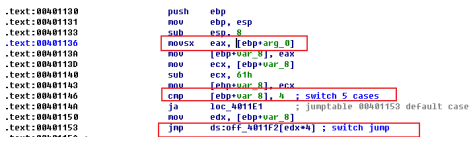


图 3.29: 0x401130 函数代码 switch_case

图3.29可以看到几个重要的参数，由此进行以下分析：

- **参数 arg_：这是最后一个被压入栈的参数，由于 x86 汇编是从右向左入栈，因此实际上就是第一个函数参数**。在这之前对应的就是下载的静态网页解析 HTML 对应的指令字符。之

后先来到 var_8，在来到 ecx，最后通过减去 0x61（ASCII 码的 a）判断是否为 0 即与 a 相等。**故实际实现的是判断函数传入的第一个参数就是 HTML 解析的指令字符是不是 a，如果是，执行命令后 ecx 归 0**。

- **switch case：**可以看到 off_4011F2 的跳转表。在运行到它之前可以看到右侧 IDA 自己显示了 **switch 5 cases**。实际上做了将 ecx 与 4 进行比较，目的是若 ecx 不为 a，判断其是否在 a 的 4 范围内，即是否为 a,b,c,d,e。**若都不是，即与 a 的距离大于 4，触发 jump above(ja)，离开这段代码到 lco_4011E1，否则会来到一个跳转表 off_4011F2。因此实际上 a-e 相当于 switch_case 语句的前五个 case，都不符合相当于 default。因此这段相当于一个 switch case**。

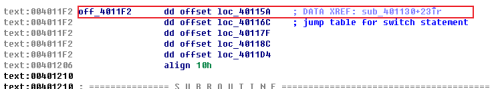


图 3.30: 跳转表

图3.30来到前面说过的跳转表，一共有五条记录，对应着 a,b,c,d,e 五个 case。

跳转表（Jump Table）是一种数据结构，编译器在为 switch 生成汇编语句时候常用。在程序中实现多路分支或多个选项的跳转。它通常是一个数组或者一个表格，其中每个元素都对应着一个特定的选项或条件，并且与每个选项或条件相关联的是一个跳转目标。

EDX 乘以 4 来到跳转表。这是因为某个分支对应的路径地址是内存地址，在 x86 中是 4 个字节。

新的函数 0x401130 包含的结构是一条 switch 语句和一个跳转表。

4. **Q4: 这个函数能够做什么?**

   回答：使用之前的 Flow Chart 查看新函数 0x401130 的流程图：

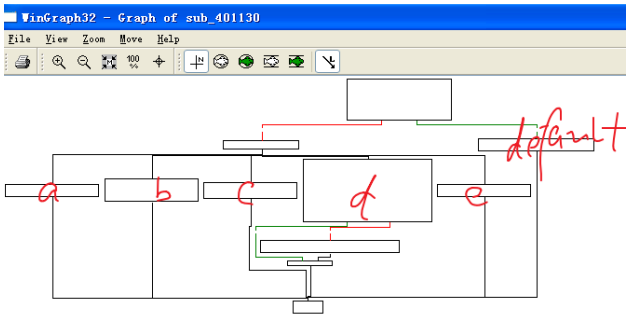
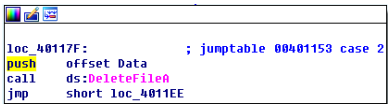


图 3.31: 0x401130 流程图

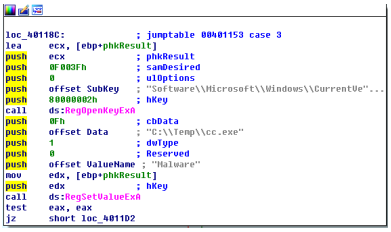
图3.31可以看到代码如之前所述，有对应于 switch 语句 5 个 case（传入的 HTML 解析字符为 a-e 的一个）和一个 default（以上都不是）一共六条可执行路径。接下来挨个进行分析：

((a)) case0:a

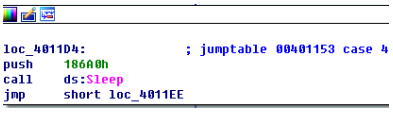
((b)) case1:b



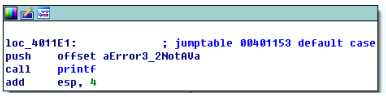
((c)) case2:c



((d)) case3:d



((e)) case4:e



((f)) default

图 3.32: 0x401130 分支功能分析

图3.32可以看到是对 0x401130 一共六个分支的不同功能代码图，接下来分别对其进行详细分析：

(a) case0(a)：使用 CreateDirectory 函数，目的是进行目录创建。C:\\Temp 作为参数，判断该目录是否存在，若不存在就**创建该目录。这为后面恶意代码其它部分的功能奠定了基础**。

(b) case1(b): 使用 CopyFile 函数，实现将某个文件复制到另一个目录下。参数分为源文件，和目标文件。

- 源文件：传递给当前函数的一个参数，即 Argv[0] 是 Lab06-03.exe。
- 目的文件：C:\Temp\cc.exe。即在刚刚创建的目录 C:\\Temp 下。

因此总体来说实现了将 Lab06-03.exe 复制为 C:\Temp\cc.exe。

(c) case2(c): 使用 DeleteFile 函数，目的是进行文件删除。C:\\Temp\\cc.exe 作为函数参数，目的是若该文件存在，则进行删除。**这是将原来的已经存在的函数删除**。

(d) case3(d): 注册表注册表键 Software\Microsoft\Windows\CurrentVersion\Run\Malware 位置处，即**在 autorun 位置创建 Malware 注册表键值，将其设置为 C:\Temp\cc.exe**。由此和之前将 Lab06-03.exe 移动到 C:\Temp，实现了恶意代码的系统启动自运行。

(e) case4(e): 压入参数 186A0h, 换算为毫秒再到秒是 100s，然后调用 Sleep。**因此会休眠 100 秒**。

(f) default case: 传用参数 aError3_NotAVa, 进一步查看其字符串**对应错误提示消息 “Error 3.2:Not a valid command provided.” 然后调用 printf 将错误信息打印**。

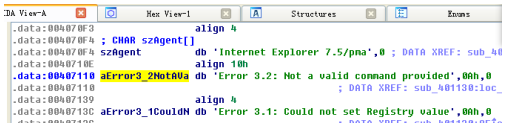


图 3.33: aError3_NotAVa

因此结合之前的分析结果可以得出，新的函数 0x401130 删除一个文件、创建一个文件夹目录、复制文件，设置一个注册表项的值完成自启动，或者休眠 100 秒等。并且还可以可以打印出错信息。总的来说，就是实现了移动到某个目录，修改注册表进行自启动，最后开始睡觉。

5. **Q5: 在这个恶意代码中有什么本地特征吗?**

   回答： 结合之前的分析，本地特征有两项：

   - 注册表键 Software\Microsoft\Windows\CurrentVersion\Run\Malware
   - 文件路径 C:\Temp\cc.exe

6. **Q6: 这个恶意代码的目的是什么?**

   回答：综合上述分析可以得出结论：

   该恶意代码先检查是否存在可以利用的有效的 Internet 连接。如果找不到，程序直接终止。否则，该程序会尝试下载一个网页，该网页包含了一段以 <!–开头的 HTML 注释。该注释的第一个字符被解析，然后利用。其具体的值决定着某种命令和分支方向。用于 switch 语句来决定程序在本地系统运行的下一步行为，包括是否创建一个目录、设置一个注册表 autorun 来进行自启动、复制一个文件、删除一个文件、或者休眠 100 秒。

3.5 Lab06-04

3.5.1 静态分析

首先进行静态分析，使用 PEView 查看其字符串，大部分还是与 Lab06-02 和 Lab06-03 完全相同，只有一个地方出现了不同：

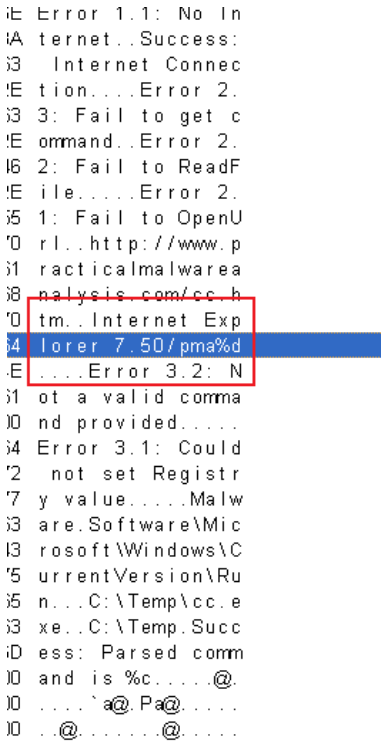


图 3.34: 查看字符串

图3.34可以看到此时多了**一个 User Agent 为 Internet Explorer 7.50/pma%d 中的%d。%d 可能是未来恶意代码的某个计算结果。它将作为函数参数未来进行动态生成 User Agent，在程序或脚本中通过代码生成一个用户代理（User Agent）字符串，而不是使用固定的静态字符串。由此作为 HTTP 请求头部的一部分。**

使用 PEiD 发现还是没有任何加壳现象，并且导入表栈中的 Windows API 函数和之前也没有新出现的，因此静态分析结束。

3.5.2 实验问题

接下来，将重点使用 IDA Pro 实现动态分析。

1. **Q1: 在实验 6-3 和 6-4 的 main 函数中的调用之间的区别是什么?**

   回答：首先还是是用 Flow Chart 查看其 main 函数的调用流程图：

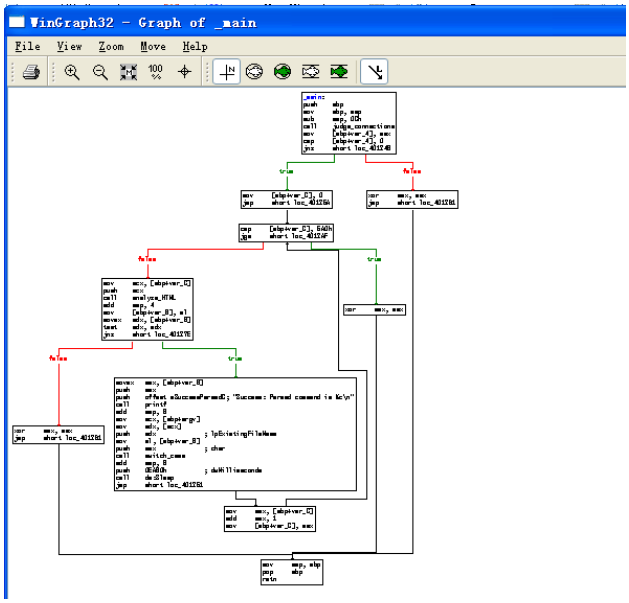


图 3.35: Lab06-04.exe 的 main 流程图

图3.35虽然图比较小，但可以放大进行仔细分析，其中再次出现了前面几个 Lab 对应的函数，**为了方便后面进行分析，这里将它们重新命名，方便理解全局功能：**

- **0x401000：judge_connections，这个地址以及功能和 6-3 完全一致**，检查是否存在 Internet 连接的函数。
- **0x401040：analyze_HTML，这个地址以及功能也和 6-3 完全一致**，下载彩蛋网页并解析 HTML 注释获取字符命令。
- **0x4012B5：printf，功能一致但地址不一致**。用来将错误信息输出。
- **0x401150：switch_case，功能一致但地址不一致**。用于根据 HTML 注释的字符命令分支进行不同功能。

因此这些函数功能都和 6-3 一致，只是结构略有不同。

2. **Q2: 什么新的代码结构已经被添加到 main 中?**

   回答：使用图形模式查看 main 流程：

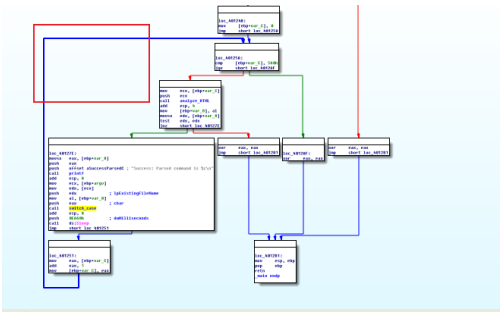


图 3.36: main 循环结构

图3.36可以看到一个向上的箭头，明显是循环结构，到相关的部分进行查看，来到循环的头部:

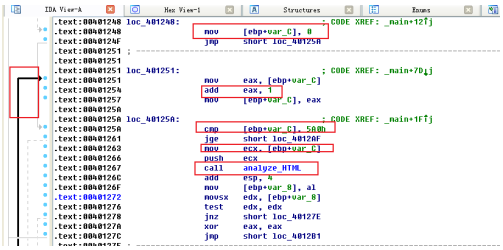


图 3.37: 循环结构

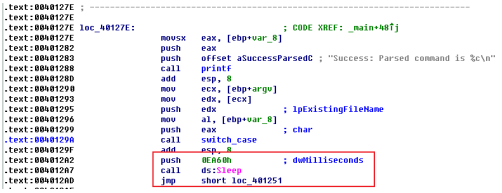


图 3.38: 循环结构 2

图3.37可以看到循环的开始位置是 **0x401248**。在此处相当于定义了一个局部变量 **var_C 用于循环计数**，首先通过 mov 命令将其赋值为 0，然后每次将其倒腾倒腾到 eax 中，然后 +1 递增，然后通过将其值和 54A0h 进行比较进行检查是否满足计数器 var_C 大于或等于 0x5A0(即 1440)，若满足通过 jge 跳转到 loc_4012AF 循环终止，否则继续执行直到末尾（即图3.38）通过 jmp short loc_401251 即递增前的位置再次循环。**由此构成了一个封闭的 for 循环结构新加入到 main 中**。。相当于如下的结构

```
for (int var_C = *(ebp+var_C); var_C <= 0x5A0; var_C++) {
    analyze_HTML(var_C);
    var_8 = al;
    int edx = var_8;
    if (edx != 0) {
        eax = 0;
        continue;
    }
    else {
```

```
10          break;
11      }
12  }
```

3. **Q3: 这个实验的解析 HTML 的函数和前面实验中的那些有什么区别?**

   回答：继续分析，发现图3.37处如果不满足计数器 var_C 大于或等于 0x5A0(即 1440)，则会执行**将 var_C 通过 ecx 压入栈上的操作，这是作为函数 analyze_HTML 的参数。但是之前的 6-2 和 6-3 中其实 analyze_HTML 并没有参数，因此一定是进行了某种更改，故重新进入其中查看：**

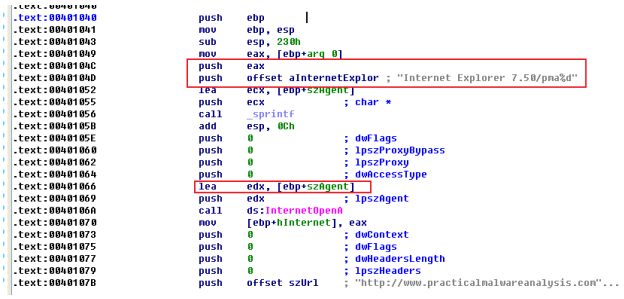


图 3.39: 查看全新的 analyze_HTML 函数

图3.39可以看到这里唯一的参事就是 **arg_0，而 arg_0 是由调用它的函数传入的，回看3.37就能发现实际上是 main 将计数器 var_C 传给了 analyze_HTML**。

然后可以看到我们在分析静态的字符串 Internet Explorer 7.50/pma%d 作为格式化字符串 (%d) 和一个目标地址一起被压入栈，供 **sprintf 调用**。其创建了一个字符串，然后将其存储在目标地址的缓冲区中。**而这里可以通过分析发现前面出现的 szAgent 就是这个缓冲区局部变量**。

在这之后，通过传入给 edx, 再让 edx 入栈，实现了**将 szAgent 作为 InternetOpenA 的函数参数**。

综上所述，0x401040 处的函数在会使用一个参数，将静态分析发现的格式化字符串 Internet Explorer 7.50/pma%d 传给 sprintf 调用，然后作为 HTTP 通信的头部成为用户代理即 User Agent 字段。

4. **Q4: 这个程序会运行多久?(假设它已经连接到互联网。)**

   回答：回到图3.38可以看到若之前的条件即计数器 var_C 大于或等于 0x5A0(即 1440) 未满足，则会继续执行循环，但是在末尾即 jmp short loc_401251 即递增前的位置再次循环前，**会通过 Sleep 休眠 600000 毫秒，即 1 分钟。因此一共当循环跳出时，var_C 已经是 1440 了，代表经历了 1440 次循环，也就是一共运行 1440 分钟，即 24 小时!**

5. **Q5: 在这个恶意代码中有什么新的基于网络的迹象吗?**

   回答：基于之前的分析，可以发现实际上 Internet Explorer 7.50/pma%d 中的%d 就是程序已经运行的循环次数，其实也是分钟数。故可作为全新的基于网络的迹象，一个全新的动态的用户代理 User Agent。

6. **Q6: 这个恶意代码的目的是什么?**

回答：综合前面所有的分析，可以发现 06-02，06-03 到 06-04 是递增的。其中最高级别的 Lab06-04.exe 有着如下的目的：

(a) 这个恶意代码首先会检查是否有可用的网络连接。如果没有找到可用的网络连接，程序将停止运行。然而，如果有可用的网络连接，程序将使用一个特殊的 User-Agent 来下载一个网页。这个 User-Agent 包含一个计数器，用于记录程序已经运行了多少分钟。

(b) 下载下来的网页中包含以 <!–开头的 HTML 注释代码。这段注释代码的第一个字符将被用于一个 switch 语句，以决定程序在本地系统中接下来的行为。这些行为可能包括硬编码的操作，比如删除一个文件、创建一个目录、设置注册表的 run 键、复制一个文件，或者让程序休眠 100 秒等。

(c) 该程序将会持续运行 24 小时后终止。

3.6 Yara 检测

3.6.1 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：

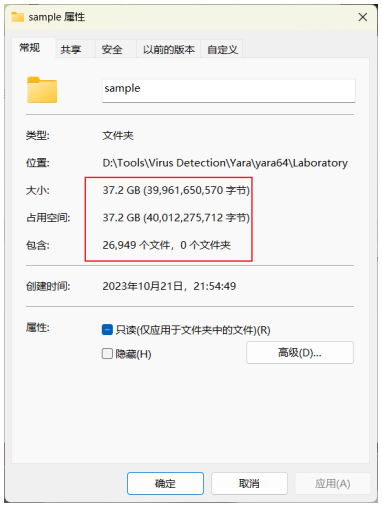


图 3.40: Sample 信息

图3.40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 37.2GB，包含一共 26949 个文件。Yara 编写的规则将目标从 sample 中识别成功检测出本次 Lab6 的全部四个恶意代码**。

3.6.2 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 字符串和动态分析结果。为了能够更好地进行 Yara 规则的编写，首先对之前分析内容进行回归。分别对四个病毒可以利用的病毒特征进行分条总结如下：

1. **Lab06-01.exe:**

   - WININET.dll: 静态分析查看函数表发现的，即网络相关的 Windows API 函数库。

- Success: Internet Connection: 通过 printf 输出的信息，用于判断是否存在可以使用的 Internet 连接并尝试连接后。
- InternetGetConnectedState: 这个 API 函数可以让应用程序通过 HTTP 协议访问 Internet 资源。

2. **Lab06-02.exe:**

- Internet Explorer: 存在于 User Agent 中的用户代理字段的部分。
- Success: Parsed command is %c：连接成功后，在命令后输入命令。解析指令的格式化字符串，也是用于通过 printf 输出在命令行中。
- http://www.practicalmalwareanalysis.com/cc.htm：恶意代码访问的 URL。

3. **Lab06-03.exe:**

- C:\\Temp\\cc.exe: 用于恶意代码将自己伪装起来的文件，是放在自己创建的目录下。
- RegSetValueExA: 恶意代码用于修改注册表键的自启动项 autorun。
- Software\\Microsoft\\Windows\\CurrentVersion\\Run：注册表自启动项 autorun 的位置，便于恶意代码定位注册表并篡改；

4. **Lab06-04.exe:**

- C:\\Temp\\cc.exe,RegSetValueExA 以及 Software\\Microsoft\\Windows\\CurrentVersion\\Run：这三项和 Lab06-03.exe 相同，因为 Lab06-04.exe 是在 03 的基础上进行的进一步改进并添加了新的功能。
- Internet Explorer 7.50/pma%d:Lab06-04.exe 的动态生成的用户代码，可以输出恶意代码已经运行的时间。

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：

```
rule Lab06_01
{
meta:
   description = "Lab06_01:Yara Rules"
   date = "2023/10/21"
   author = "ErwinZhou"
strings:
   $clue1 ="WININET.dll" wide ascii
   $clue2 ="Success: Internet Connection" wide ascii nocase
   $clue3 = "InternetGetConnectedState" wide ascii nocase

condition:
   all of them //Lab06-01.exe
}


```

```
rule Lab06_02
{
meta:
   description = "Lab6_02:Yara Rules"
   date = "2023/10/21"
   author = "ErwinZhou"

strings:
   $clue1 = "Internet Explorer" wide ascii
   $clue2 = "Success: Parsed command is %c" wide ascii nocase
   $clue3 = "http://www.practicalmalwareanalysis.com/cc.htm" wide ascii

condition:
   all of them //Lab06-02.exe
}


rule Lab06_03
{
meta:
   description = "Lab6_03:Yara Rules"
   date = "2023/10/21"
   author = "ErwinZhou"

strings:
   $clue1 = "C:\\Temp\\cc.exe" wide ascii nocase
   $clue2 = "RegSetValueExA" wide ascii nocase
   $clue3 = "Software\\Microsoft\\Windows\\CurrentVersion\\Run" wide ascii nocase

condition:
   all of them //Lab06-03.exe
}


rule Lab06_04
{
meta:
   description = "Lab6_04:Yara Rules"
   date = "2023/10/21"
   author = "ErwinZhou"

strings:
```

```
    $clue1 = "C:\\Temp\\cc.exe" wide ascii nocase
    $clue2 = "RegSetValueExA" wide ascii nocase
    $clue3 = "Software\\Microsoft\\Windows\\CurrentVersion\\Run" wide ascii nocase
    $clue4 = "Internet Explorer 7.50/pma%d" wide ascii nocase
condition:
    all of them //Lab06-04.exe
}
```

然后使用如下 Python 代码进行对 sample 的扫描:

```python
import os
import yara
import time

# 加载YARA规则
rules = yara.compile('D:\Tools\Virus Detection\Yara\yara64\Lab6.yar')

# 初始化计数器
total_files_scanned = 0
total_files_matched = 0

def scan_folder(folder_path):
    global total_files_scanned
    global total_files_matched

    # 检查文件夹是否存在
    if os.path.exists(folder_path) and os.path.isdir(folder_path):
        # 遍历文件夹内的文件和子文件夹
        for root, dirs, files in os.walk(folder_path):
            for filename in files:
                total_files_scanned += 1
                file_path = os.path.join(root, filename)
                with open(file_path, 'rb') as file:
                    data = file.read()
                    # 扫描数据
                    matches = rules.match(data=data)
                    # 处理匹配结果
                    if matches:
                        total_files_matched += 1
                        print(f"File '{filename}' in path '{root}' matched YARA
                            rule(s):")
                        for match in matches:
                            print(f"Rule: {match.rule}")
```

```
    else:
        print(f'The folder at {folder_path} does not exist or is not a folder.')

# 文件夹路径
folder_path = 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sample'
# 记录开始时间
start_time = time.time()
# 递归地扫描文件夹
scan_folder(folder_path)
# 记录结束时间
end_time = time.time()
# 计算运行时间
runtime = end_time - start_time

print(f"Program runtime: {runtime} seconds.")
print(f"Total files scanned: {total_files_scanned}")
print(f"Total files matched: {total_files_matched}")
```

3.6.3 Yara 规则执行效率测试

扫描结果如下图所示：

```
File 'Lab06-01.exe' in path 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sampl
e' matched YARA rule(s):
Rule: Lab06_01
File 'Lab06-02.exe' in path 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sampl
e' matched YARA rule(s):
Rule: Lab06_01
Rule: Lab06_02
File 'Lab06-03.exe' in path 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sampl
e' matched YARA rule(s):
Rule: Lab06_01
Rule: Lab06_02
Rule: Lab06_03
File 'Lab06-04.exe' in path 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sampl
e' matched YARA rule(s):
Rule: Lab06_01
Rule: Lab06_02
Rule: Lab06_03
Rule: Lab06_04
Program runtime: 138.41580605506897 seconds.
Total files scanned: 26949
Total files matched: 4
```

图 3.41: Yara 检测结果

图3.41可以看到**能够唯一地从 26949 个文件中识别检测到四个病毒。并且还可以额外发现越往后的病毒其还能够匹配前面的规则，这其实也是因为后面的病毒是在前面的病毒的特征上进行了拓展，检测结果也验证了这一点。**

并且用时 137.80 秒，时间性能较快。总的来说，Yara 规则编写和检测较为成功。

3.7 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：

3.7.1 方便查看函数

目的是获取一块区域地址在的函数的起始和结束地址。

```
# 获取当前选中的地址
ea = here()
# 使用 IDA API 获取该地址所在的函数信息
func = idaapi.get_func(ea)
# 打印函数的起始和结束地址
print "Start: 0x%x, End 0x%x" % (func.startEA, func.endEA)
```

3.7.2 反汇编

获取一个地址所在的函数的起始和结束地址，并遍历该函数内的每一个指令，打印出每个指令的地址和反汇编代码。

```
# 获取当前选中的地址
ea = here()
# 使用 IDA API 获取该地址所在的函数的起始地址
start = idc.get_func_attr(ea, FUNCATTR_START)
# 使用 IDA API 获取该地址所在的函数的结束地址
end = idc.get_func_attr(ea, FUNCATTR_END)
# 设置当前地址为函数的起始地址
cur_addr = start
# 遍历函数内的每一个指令
while cur_addr <= end:
    # 打印指令的地址和反汇编代码
    print hex(cur_addr), idc.generate_disasm_line(cur_addr, 0)
    # 获取下一个指令的地址
    cur_addr = idc.next_head(cur_addr, end)
```

3.7.3 识别库函数

遍历所有的函数，并检查每个函数是否被标记为库函数（FUNC_LIB）。对于被标记为库函数的，打印出函数的地址、标记和函数名。

```
# 使用 IDA API 遍历所有的函数
for func in idautils.Functions():
    # 获取函数的属性标记
    flags = idc.get_func_attr(func, FUNCATTR_FLAGS)

    # 检查函数是否被标记为库函数
    if flags & FUNC_LIB:
        # 打印函数的地址、标记和函数名
```

```
9         print hex(func), "FUNC_LIB", idc.get_func_name(func)
```

3.7.4 函数 jup 跳转与 call 调用

遍历所有的函数，并对每个非库函数和非跳转函数（非 FUNC_LIB 和非 FUNC_THUNK）进行进一步的分析。查找函数内的所有 call 和 jmp 指令，并进一步检查这些指令的操作数是否是寄存器。对于满足条件的指令，打印出指令的地址和反汇编代码。

```
# 使用 IDA API 遍历所有的函数
for func in idautils.Functions():
    # 获取函数的属性标记
    flags = idc.get_func_attr(func, FUNCATTR_FLAGS)
    # 检查函数是否被标记为库函数或跳转函数
    if flags & FUNC_LIB or flags & FUNC_THUNK:
        continue # 如果是，则跳过该函数
    # 获取函数内的所有指令地址
    dism_addr = list(idautils.FuncItems(func))
    # 遍历函数内的每一个指令地址
    for line in dism_addr:
        # 获取指令的助记符（例如 'call', 'jmp' 等）
        m = idc.print_insn_mnem(line)
        # 检查指令是否是 'call' 或 'jmp'
        if m == 'call' or m == 'jmp':
            # 获取指令的第一个操作数的类型
            op = idc.get_operand_type(line, 0)
            # 检查操作数是否是寄存器
            if op == o_reg:
                # 打印指令的地址和反汇编代码
                print "0x%x %s" % (line, idc.generate_disasm_line(line, 0))
```

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 动态分析的结果相同，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 Lab6 的四个恶意代码进行了全面的分析，并依次回答了书中的问题。并在通过分析汇编结构的过程中，识别出了许多独特的 C 语言结构。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动态分析工具 IDAPro，用于更全面的动态分析；
2. 其次我通过观察 IDA 显示的汇编识别 C 语言结构并分析其目的；
3. 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。
4. 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，和通过汇编语言识别 C 结构的能力。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:)