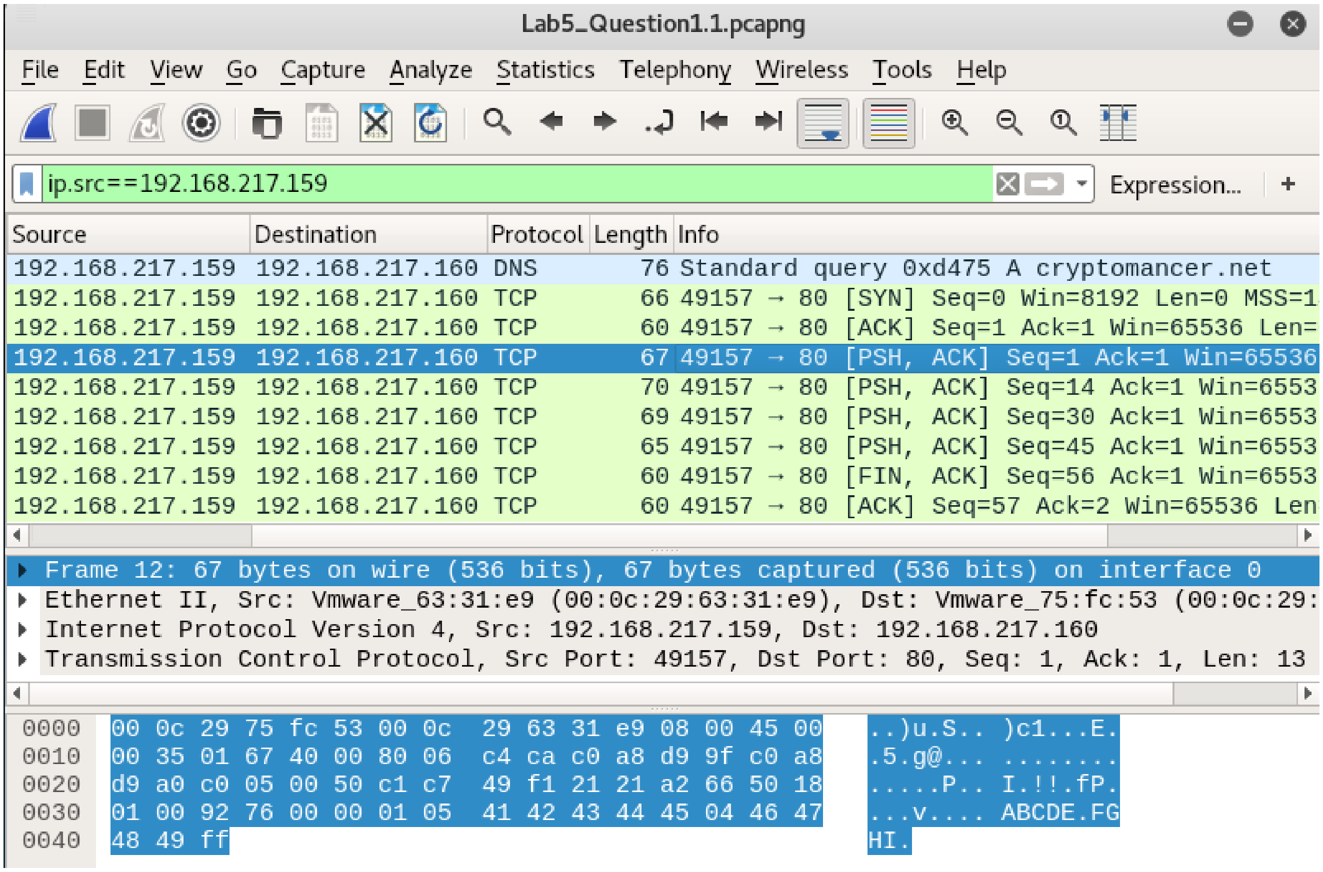
ISA 564: Lab 5 – Network Hunting and C2 Answers

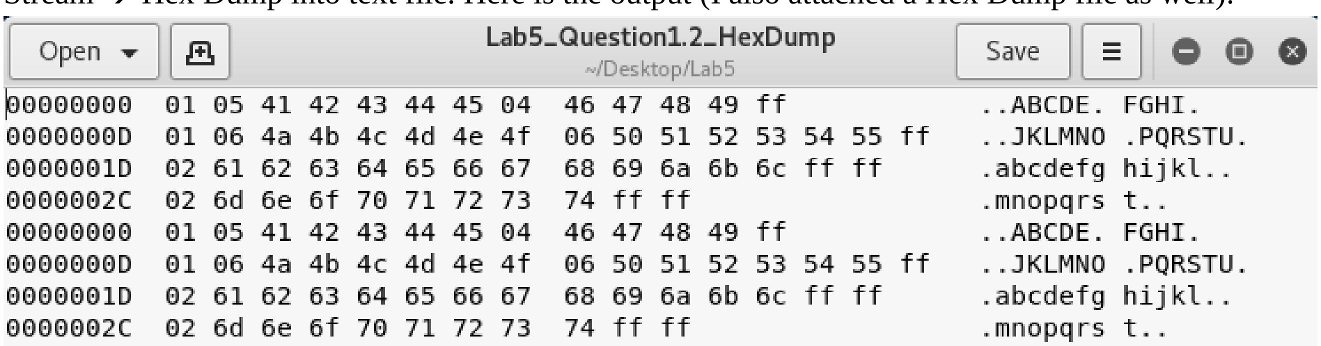
(by Sattyik Kundu)

I. Task 1 Answers:

Question 1.1 – I have attached a PCAP file from a Wireshark scan showing the successful beaconing of the malware to the Kali VM (running INetsim). In the below screenshot, I filtered the capture file so only the traffic going from Windows VM(source) → Kali VM is shown. This is to demonstrate how the Windows VM successfully beacons and establishes a C2 session with Kali VM; since the session SYN and ACK packets started being passed inbetween the moment I clicked onto Lab5.1.exe in Windows VM:



Question 1.2 – I right-clicked the same TCP packet that I highlighted above and then I selected Follow → TCP Stream → Hex Dump into text file. Here is the output (I also attached a Hex Dump file as well):



The relevant sections of code on each line starts with "01" or "02"; and then ends with "ff" or "ff ff", respectively. Additionally, ONLY the 1st four lines are unique. The 2nd four lines are just a repeat of the first four. Here are the annotations for the 4 lines:

For the 1st line (“01 05 41 42 43 44 45 04 46 47 48 49 ff”):

I am pretty sure the “05” refers the field of 5 bytes that comes afterwards; this 5-bytes data is “41 42 43 44 45”. This means that subsequent “04” refers to the following 4-byte field of “46 47 48 49”. The “ff” byte at the end may mean an ending flag. As to what the functions of the 6-byte and 3-byte fields are, more comprehensive code analysis will be needed.

For the 2nd line (“01 06 4a 4b 4c 4d 4e 4f 06 50 51 52 53 54 55 ff”):

Here, there are two “06” values. This means that there are two data segments of 6-bytes each. The 1st 6-byte data consists of “4a 4b 4c 4d 4e 4f”. The next 6-byte field consists of “50 51 52 53 54 55” following the 2nd “06”. Lastly, like before, the “ff” can be the ending flag. Further code analysis is needed to determine what the functions of the two 6-byte fields are.

For the 3rd line (“02 61 62 63 64 65 66 67 68 69 6a 6b 6c ff ff”):

This time, there are no bytes to represent field sizes. Between “02” and “ff ff”, there is one set of consecutive hex values of “61 62 63 64 65 66 67 68 69 6a 6b 6c”. Even though there are no bytes to define field sizes (like in the previous 2 lines), the ending flag of “ff” can instead be used to determine when the byte pattern ends.

Additionally, it is now possible to believe that the “02” at the beginning refers to the two “ff” end flags at the end. In the previous 1st and 2nd lines, they each started with “01” and ended with only one “ff”; which means that “01” refers to having only one “ff” flag. More comprehensive code analysis will be needed to determine this line’s meaning.

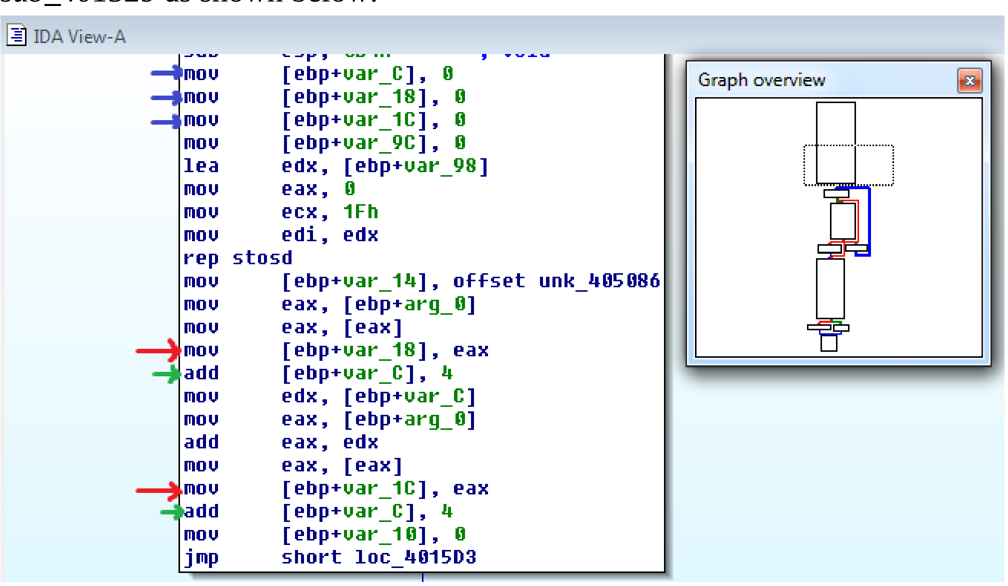
For the 4th and last line (“02 6d 6e 6f 70 71 72 73 74 ff ff”):

Just like the previous(3rd) line, it starts with “02” which means that the entire hex segment ends with two “ff” ending flags. Similar to before, the entire segment between “02” and “ff ff” consist of consecutive hex values (which is “6d 6e 6f 70 71 72 73 74”). Also like in the 3rd line, there are no hex values determining field sizes. With no field sizes defined, the byte pattern’s start and end can be interpreted as starting from “02” and ending at “ff” followed by “ff”.

II. Task 2 Answers:

Question 2.1 –

To start with, the first 8 bytes of the input are already read through and stored within the 1st code block of *sub_401529* as shown below:



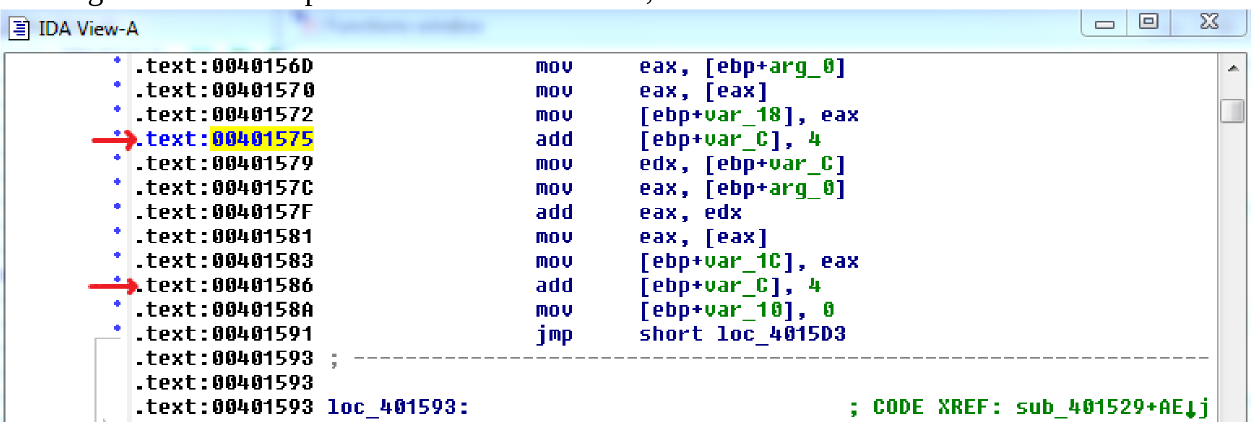
In the 1st three instructions above (with blue arrows), the variables of var_C, var_18, and var_1C are all initialized to 0.

Later down at instruction “mov [ebp+var_18], eax” where 1st red arrow points, the first 4 bytes (32 bits) of the input that was stored in register eax is moved to [ebp+var_18]. In the following line of “add [ebp+var_C], 4”, 4 is added to [ebp+var_C] because the var_C variable **represents** the number of bytes read so far. That means 4 bytes have been read so far since it was earlier initialized to 0.

Going down further, at “mov [ebp+var_1C], eax” where the 2nd red arrow points, the next 4 bytes of the input stored in eax is moved to [ebp+var_1C]. Afterwards, in “add [ebp+var_C], 4”, 4 is again added to [ebp+var_C]. Now, var_C is 8 since 8 bytes have been read and stored so far.

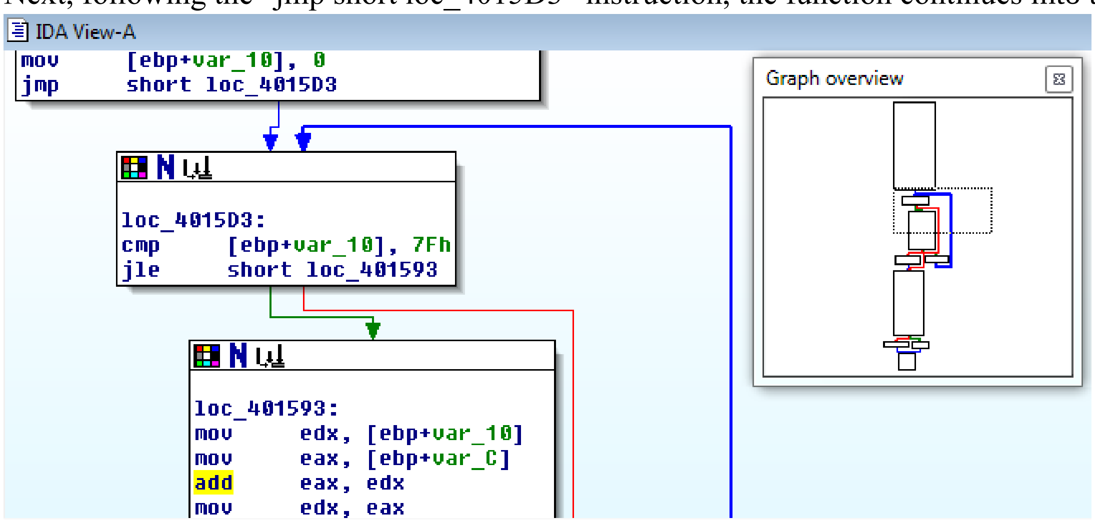
As the first and second 4 bytes of the input have been moved to var_18 and var_1C, respectively; what remains of the input bytes that can be read will start from the 9th byte.

The lab rubric asks for the addresses of the instructions where the numbers of bytes read are updated. By right-clicking the current Graph view →click Text view, the addresses of the instructions are shown:



As shown in the above text-view of function sub_401529, the addresses of the 1st and 2nd “add [ebp+var_C], 4” instructions are 00401575 and 00401586, respectively.

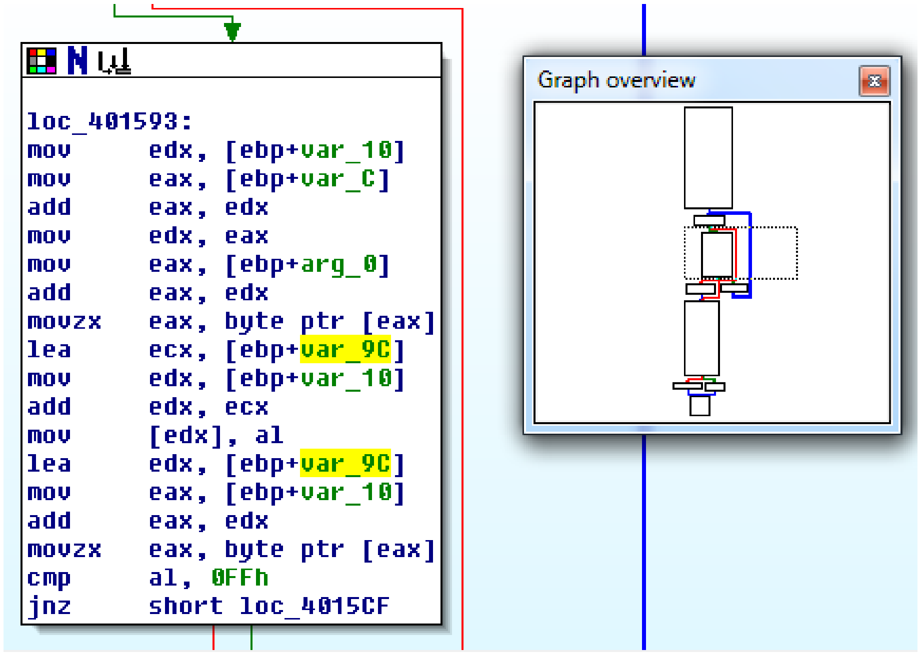
Next, following the “jmp short loc_4015D3” instruction, the function continues into the next code segment:



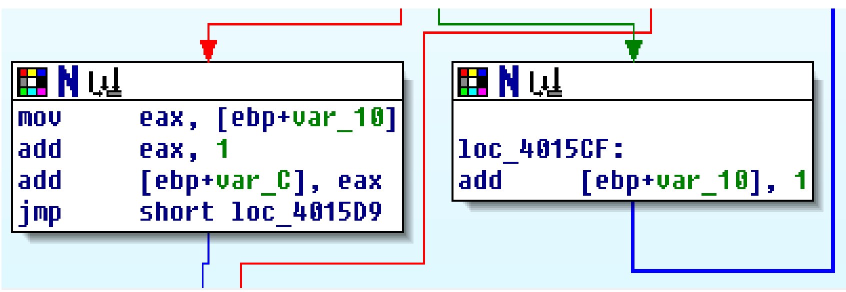
After var_10 is initialized to zero in “mov [ebp+var_10], 0”, the program jumps to loc_4015D3. There lies the conditions for executing a loop in this function. The “cmp [ebp+var_10], 7Fh” instruction compares the var_10 value to 7Fh (“h” stands for hex-value; in decimal form, 7F converts to 127).

In the following function of “jle short loc_401593”, if [ebp+var_10] is less than or equal to 7Fh (or 127 in decimal), the execution jumps to loc_401593 (as shown above and below).

Here is the rest of loc_401593 data block in the loop:



When [ebp+var_10] is initially zero, the loc_401593 data block will start reading from the 9th byte of the original input (since the first 8 bytes have already been read and moved). Eventually, the program continues down to instruction “cmp al, 0FFh”. The register al, which holds the current byte (which is the 9th byte of the input when [ebp+var_10] equals zero), is checked to see if it equals FF (which is in hex-value). If not, the program jumps to loc_4015CF (as shown in below screenshot which also shows the rest of the loop).

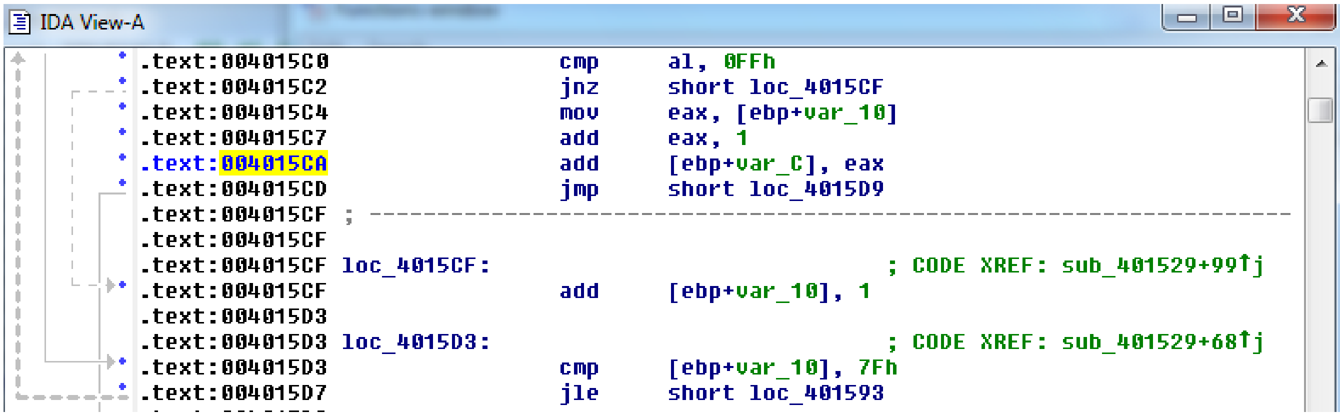


At loc_4015CF, the instruction “add [ebp+var_10], 1” increments var_10 by 1. This is done because var_10 represents the number of bytes read so far. Since the 9th byte of the input was read during the first iteration of the loc_401593 data block, var_10 increments by 1.

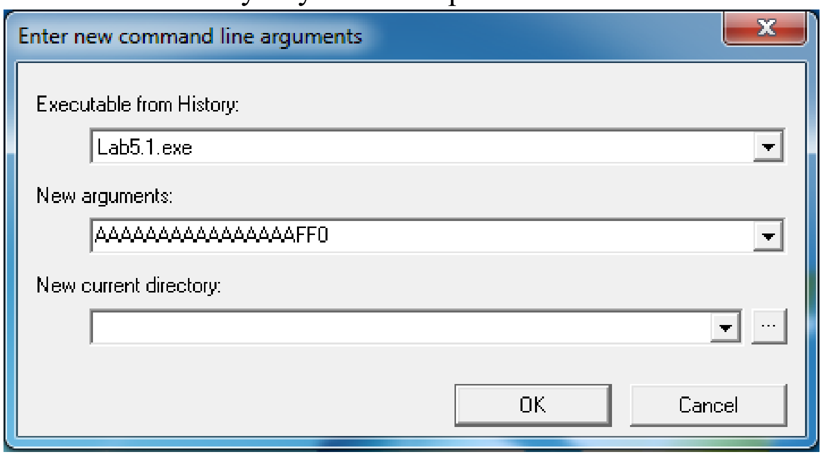
The program loops back to loc_4015D3. Unless [ebp+var_10] is greater than 7Fh, it should continue into loc_401593. Now, during the 2nd iteration of going through the loc_401593 data block, the “cmp al, 0FFh” instruction will check if the input’s 10th byte equals “FF”. If not, the loop will keep iterating though each of the remaining bytes of the input until a byte of “FF” is found in register al. If register al has the value FF, it will proceed to the code block in the above bottom-left.

In the above bottom-left code block, the [ebp+var_10] of instruction “mov eax, [ebp+var_10]” represents the number of bytes read after the input’s 8th byte as a result of iterations in the loop; minus the byte read in the last iteration (due to not going to loc_4015CF to increment var_10). The next instruction corrects this number stored in register eax by using instruction “add eax, 1”. In the 3rd instruction, which is “add [ebp+var_C], eax”, the total number of bytes read during the loop iterations is finally added to the 8 bytes read before the loop. This results in [ebp+var_C] finally having the total number of the input bytes being read before the program completion. The var_C shouldn’t change anymore afterwards as function sub_401529 heads to completion.

Finally, with "add [ebp+var_C], eax" being the 3rd and last instruction which updates the number of bytes read, the instruction's address is found to be 004015CA (as shown below using text-view):

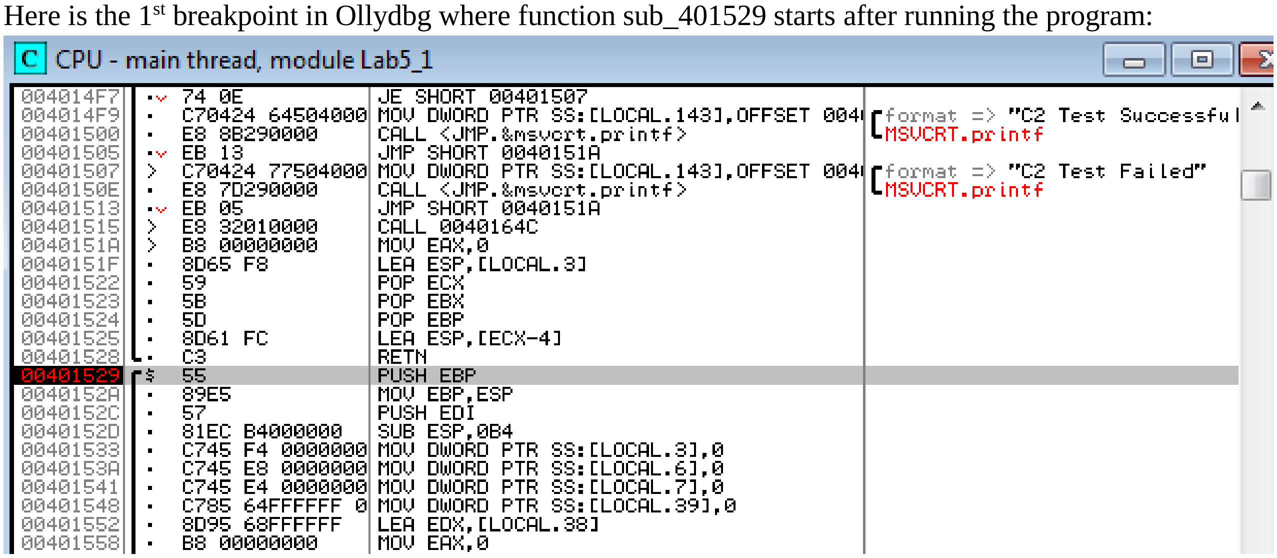


Question 2.2 – Now, binary Lab5.1.exe needs to be executed in Ollydbg using an input argument what would enable to the binary to yield an output of "C2 Test Succeeded". The output I have chosen is as shown below:

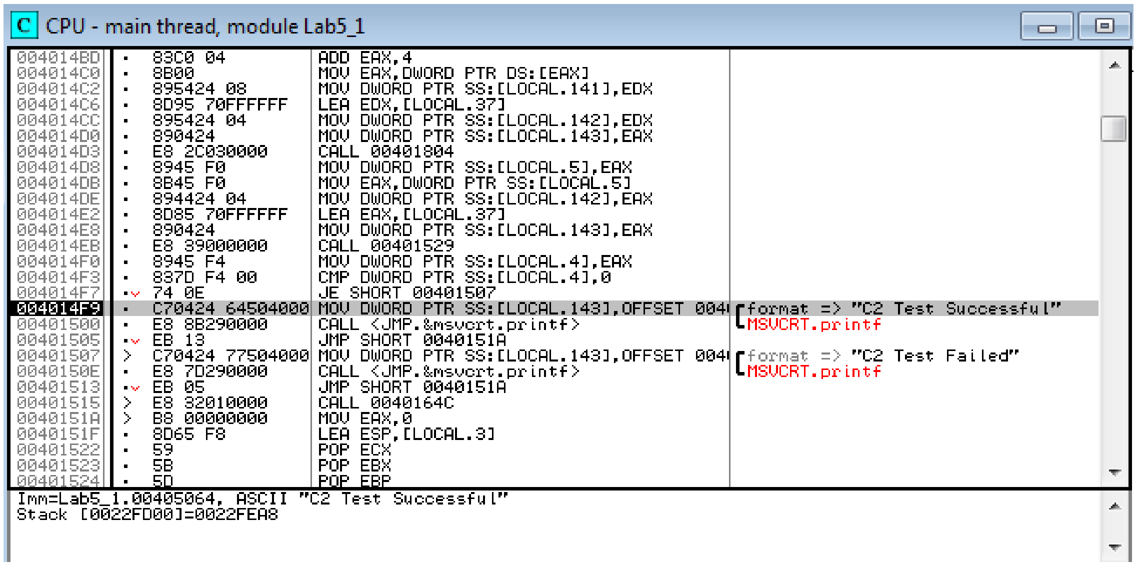


The input consists of 16 "A" hex values followed by "FF0". In Question 2.1, I have stated that "FF" needed to be placed after the 1st 8 bytes to be successfully read by the "cmp al, 0FFh" instruction explained earlier. Two hex-values make one byte because hex values are 4 bits each; hence, the 16 hex "A" values will make a total of 8 bytes needed. Then the "FF" appended afterwards can then be successfully read by "cmp al, 0FFh" within the loop.

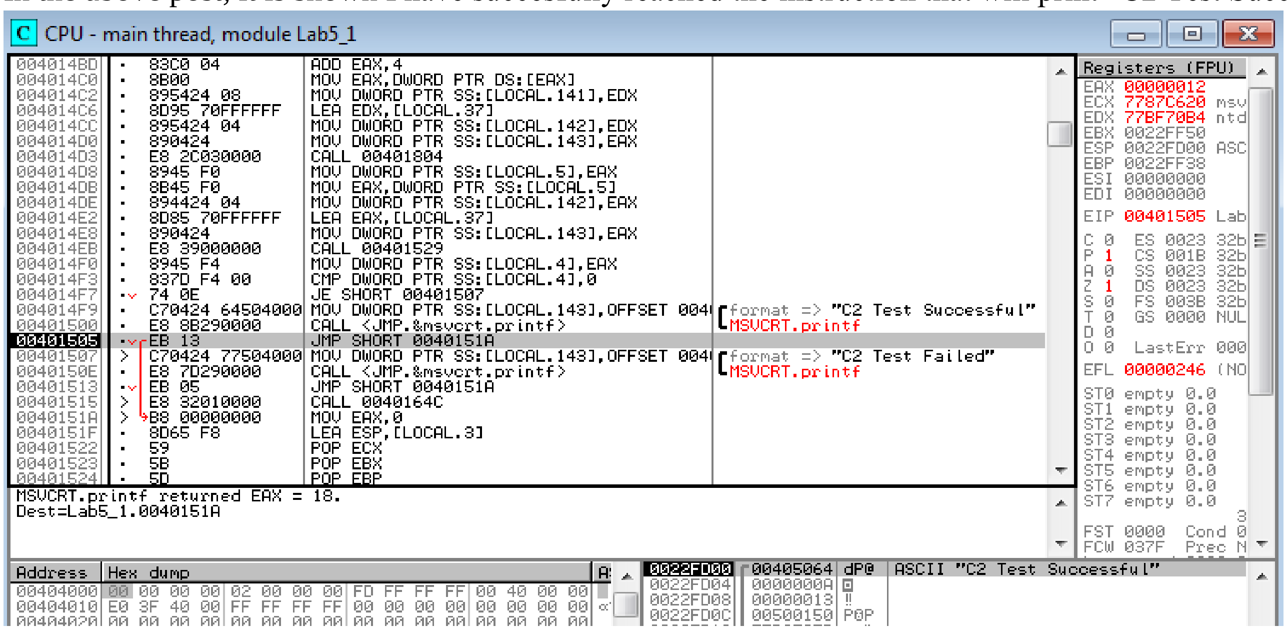
Here is the 1st breakpoint in Ollydbg where function sub_401529 starts after running the program:



Now, I step over the function until I reach the instruction "cmp al, 0FFh" where al=FF.
As seen above, I successfully execute instruction "cmp al, 0FFh". In the bottom left of the above screenshot, it shows that AL=FF. From then on, the program should continue until "C2 Test Succeeded" is successfully printed.



In the above post, it is shown I have succesfully reached the instruction that will print “C2 Test Successful”.

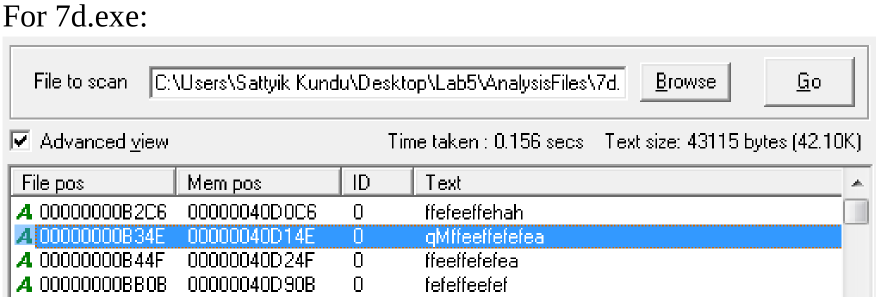


Finally, after stepping over the “CALL <JMP.&msvcrt.printf>” instruction, the program prints in ASCII “C2 Test Successful” in the bottom-right output pane (as shown above-right). With this, it can be said the binary execution was successfully given the input. By the way, the reason I added zero at the end of my input was to get the program to not jump past the “C2 Test Successful” print string even though al=FF even without the zero.

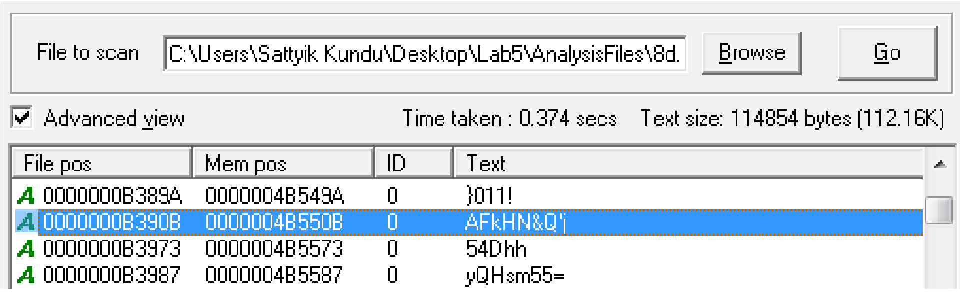
<u>III. Task 3 Answers:</u>

Question 3.1 – I am reusing Bintext (from Lab 4) in Windows to look for unique strings for the 5 malwares. I will try to avoid gibberish strings in favor of more comprehensible strings for ease. Below are the 5 unique strings via Bintext I am using:

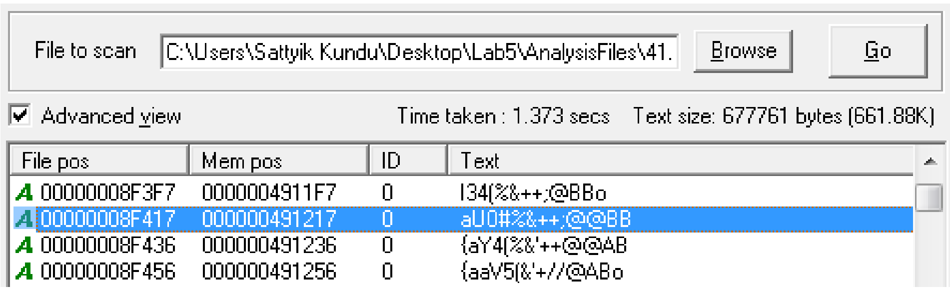
For 7d.exe:



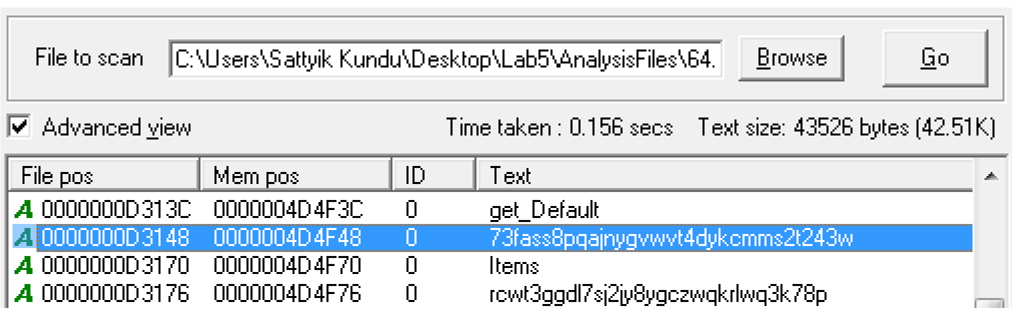
For 8d.exe:



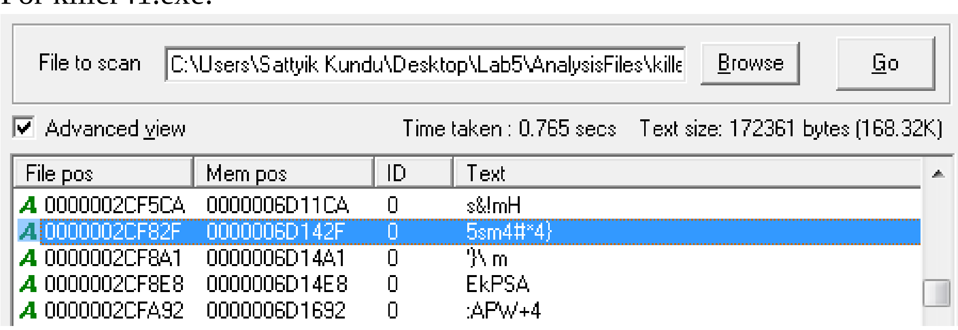
For 41.exe:



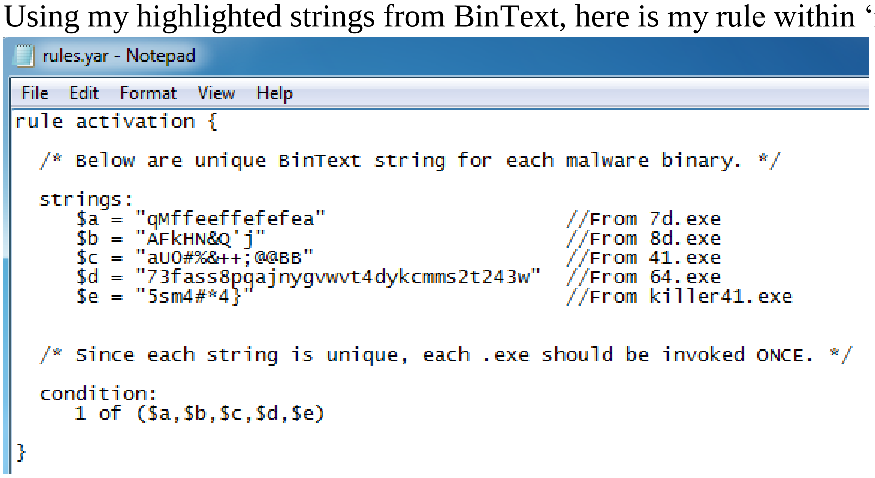
For 64.exe:



For killer41.exe:

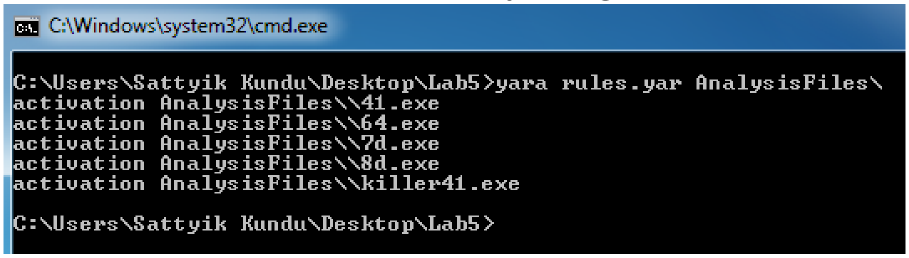


Using my highlighted strings from BinText, here is my rule within ‘rules.yar’ file(attached to lab):




```
rule activation {

  /* Below are unique BinText string for each malware binary. */

  strings:
     $a = "qMffeeffefefea"                    //From 7d.exe
     $b = "AFkHN&Q'j"                         //From 8d.exe
     $c = "aU0#%&++;@@BB"                     //From 41.exe
     $d = "73fass8pqajnygvwvt4dykcmms2t243w"  //From 64.exe
     $e = "5sm4#*4}"                          //From killer41.exe


  /* Since each string is unique, each .exe should be invoked ONCE. */

  condition:
     1 of ($a,$b,$c,$d,$e)
}
```

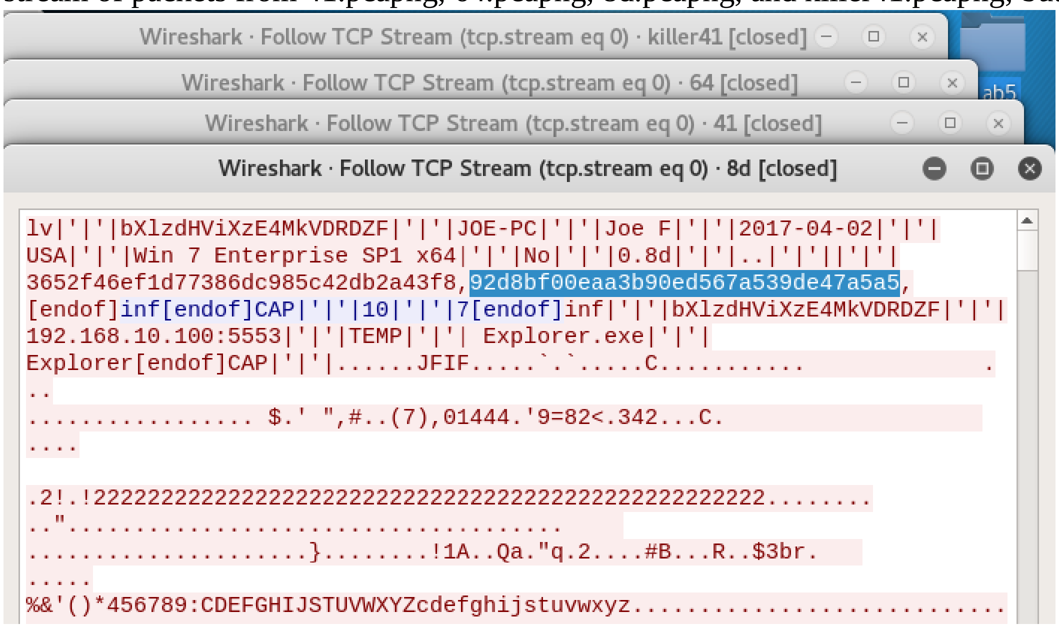
Now here is a screenshot of successfully hitting all 5 malware binaries using a YARA rule:



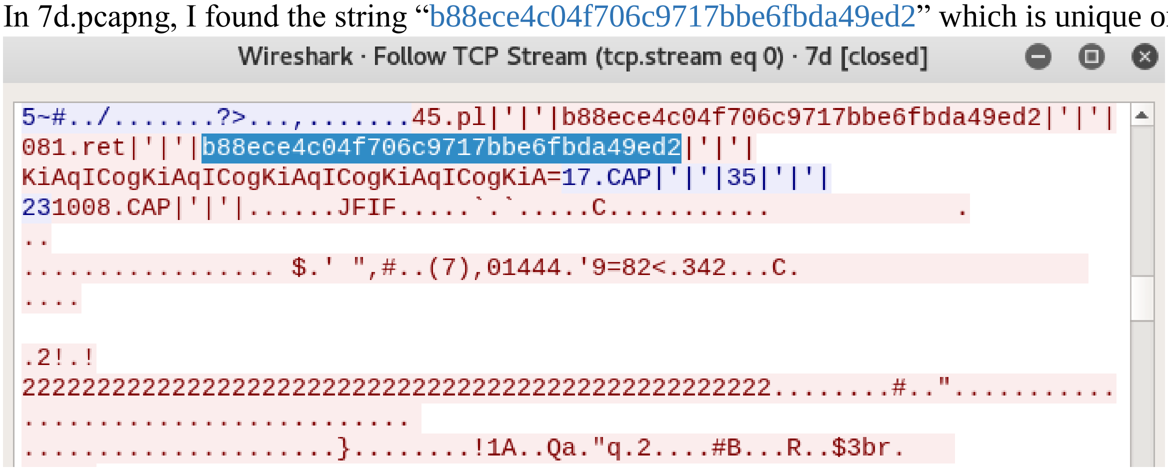
IV. Task 4 Answers:

Question 4.1 – In Wireshark, I opened each PCAP file. For various packets, I right-clicked and then selected Follow → TCP Stream → ASCII find a common string pattern(s) that can be used to identify all the PCAP files **except** for *tcp.pcapng*. Additionally, the rules stated that the string CANNOT include hostnames, IP addresses, OS identifiers, and filenames.

During my initial search, I found the string "92d8bf00eaa3b90ed567a539de47a5a5" to be common in the TCP stream of packets from 41.pcapng, 64.pcapng, 8d.pcapng, and killer41.pcapng; but not in 7d.pcapng.



In 7d.pcapng, I found the string "b88ece4c04f706c9717bbe6fbda49ed2" which is unique only to this PCAP file:



In the end, I was unable to find a common string for all five njRAT PCAP files. So I decided to create a snort rule that would determine if any of the files had either the "92d8bf00eaa3b90ed567a539de47a5a5" or "b88ece4c04f706c9717bbe6fbda49ed2" strings to cover all 5 njRAT PCAP files when searching all packets.

Hence, here is my snort rule:

local.rules
/etc/snort/rules

```
# $Id: local.rules,v 1.11 2004/07/23 20:15:44 bmc Exp $
# ----------------
# LOCAL RULES
# ----------------
# This file intentionally does not come with signatures.  Put your local
# additions here.

alert tcp any any -> any any (pcre:"/92d8bf00eaa3b90ed567a539de47a5a5|
b88ece4c04f706c9717bbe6fbda49ed2/"; sid:1000000;)
```

In the above rule, the pcre (Pearl compatible regular expression) checks for a string match for "92d8bf00eaa3b90ed567a539de47a5a5" **OR** "b88ece4c04f706c9717bbe6fbda49ed2".

Finally, here is my snort rule execution output:

```
root@kali:~/Desktop/Lab5# snort -c /etc/snort/snort.conf -q -k none -A console --pcap-s
how --pcap-dir AnalysisPCAPs
Reading network traffic from "AnalysisPCAPs/41.pcapng" with snaplen = 1514
04/02-20:38:06.258153  [**] [1:1000000:0] [**] [Priority: 0] {TCP} 192.168.10.100:1177
-> 192.168.10.100:1030
04/02-20:38:06.270931  [**] [1:1000000:0] [**] [Priority: 0] {TCP} 192.168.10.100:1030
-> 192.168.10.100:1177
Reading network traffic from "AnalysisPCAPs/64.pcapng" with snaplen = 1514
04/02-20:39:54.732878  [**] [1:1000000:0] [**] [Priority: 0] {TCP} 192.168.10.100:1033
-> 192.168.10.100:5555
Reading network traffic from "AnalysisPCAPs/7d.pcapng" with snaplen = 1514
04/02-20:46:03.434015  [**] [1:1000000:0] [**] [Priority: 0] {TCP} 192.168.10.100:4000
-> 192.168.10.100:1041
04/02-20:46:03.439431  [**] [1:1000000:0] [**] [Priority: 0] {TCP} 192.168.10.100:1041
-> 192.168.10.100:4000
04/02-20:46:03.491127  [**] [1:1000000:0] [**] [Priority: 0] {TCP} 192.168.10.100:1041
-> 192.168.10.100:4000
Reading network traffic from "AnalysisPCAPs/8d.pcapng" with snaplen = 1514
04/02-20:43:08.130909  [**] [1:1000000:0] [**] [Priority: 0] {TCP} 192.168.10.100:1039
-> 192.168.10.100:5553
04/02-20:43:47.230968  [**] [1:1000000:0] [**] [Priority: 0] {TCP} 192.168.10.100:5553
-> 192.168.10.100:1039
04/02-20:43:47.233488  [**] [1:1000000:0] [**] [Priority: 0] {TCP} 192.168.10.100:1039
-> 192.168.10.100:5553
Reading network traffic from "AnalysisPCAPs/killer41.pcapng" with snaplen = 1514
04/02-20:48:35.671637  [**] [1:1000000:0] [**] [Priority: 0] {TCP} 192.168.10.100:1045
-> 192.168.10.100:6661
Reading network traffic from "AnalysisPCAPs/tcp.pcapng" with snaplen = 1514
root@kali:~/Desktop/Lab5#
```

As seen above, my snort alert rule has successfully read traffic from all 5 of the njRAT PCAP files (underlined in orange using Paint). Additionally, there is no traffic being read from the tcp.pcapng file (underlined in red) as intended.