Dynamic Analysis

Scenario: Many years ago, in the depths of the sands of Egypt lied a great

Pharaoh. One day his servant betrayed him and left a curse across the land. Buried under the pyramids, his corpse lay dormant, until one day; an adventurer uncovered his tomb and unleashed chaos into the world. The mummy has awakened and is ravaging the city.

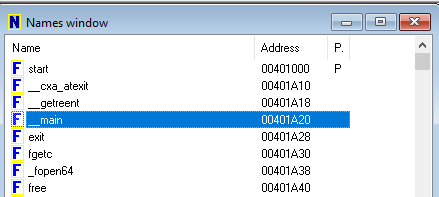
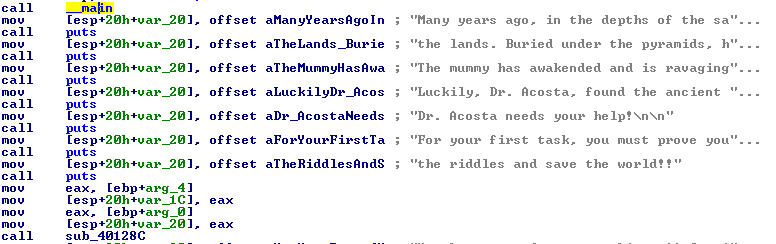
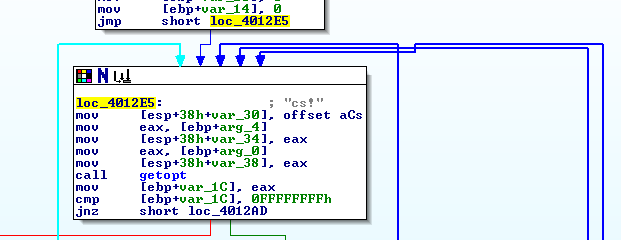
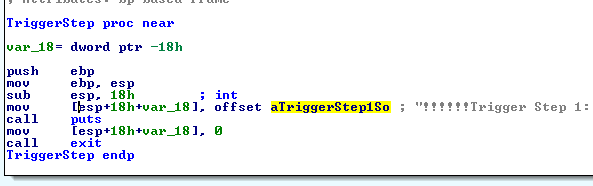
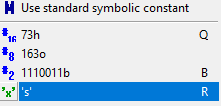
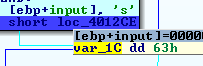
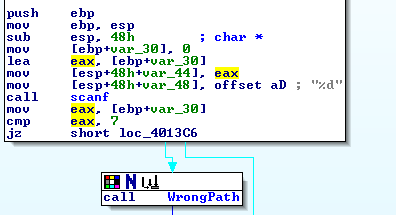
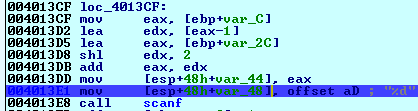
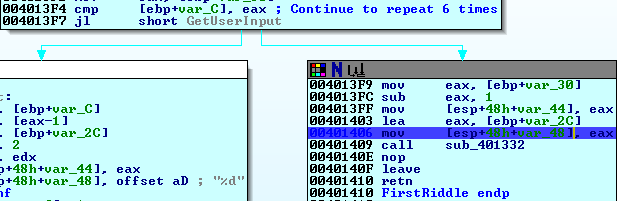
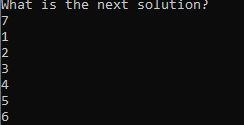
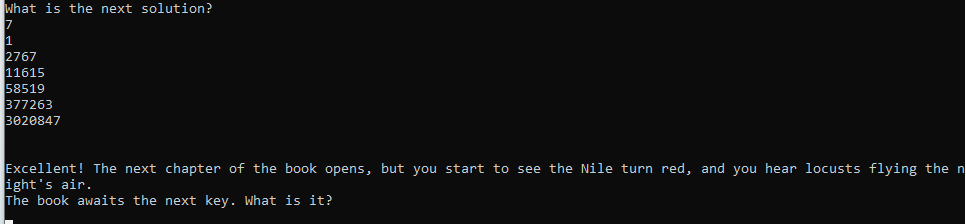
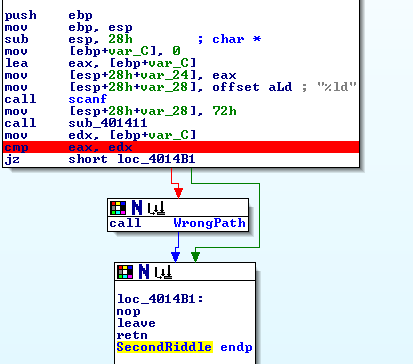
Luckily, Acosta, found the ancient "book of the dead" (x86 binary) that has the key to take down the mummy!

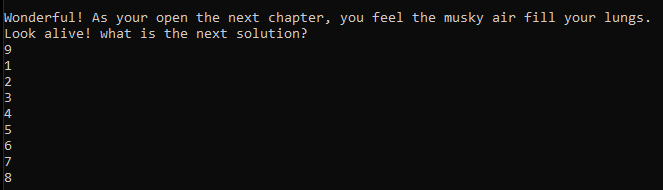
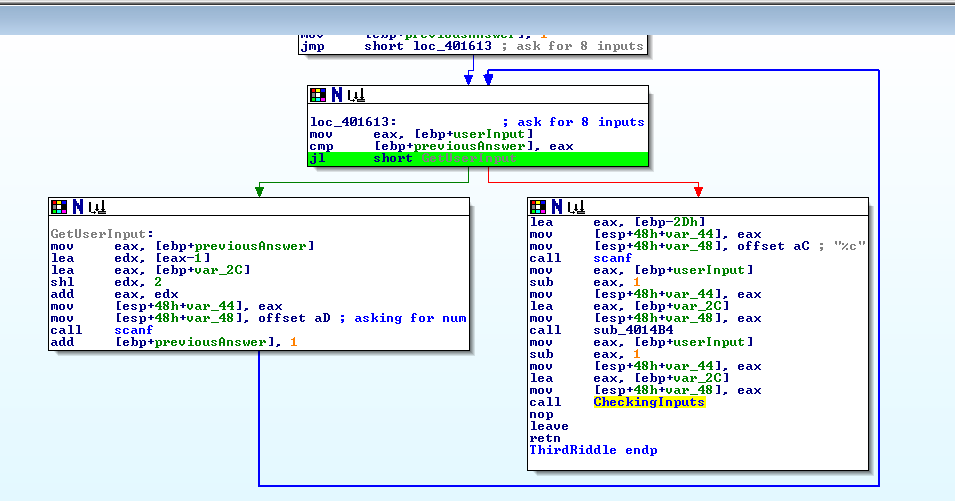
Dr. Acosta needs your help!

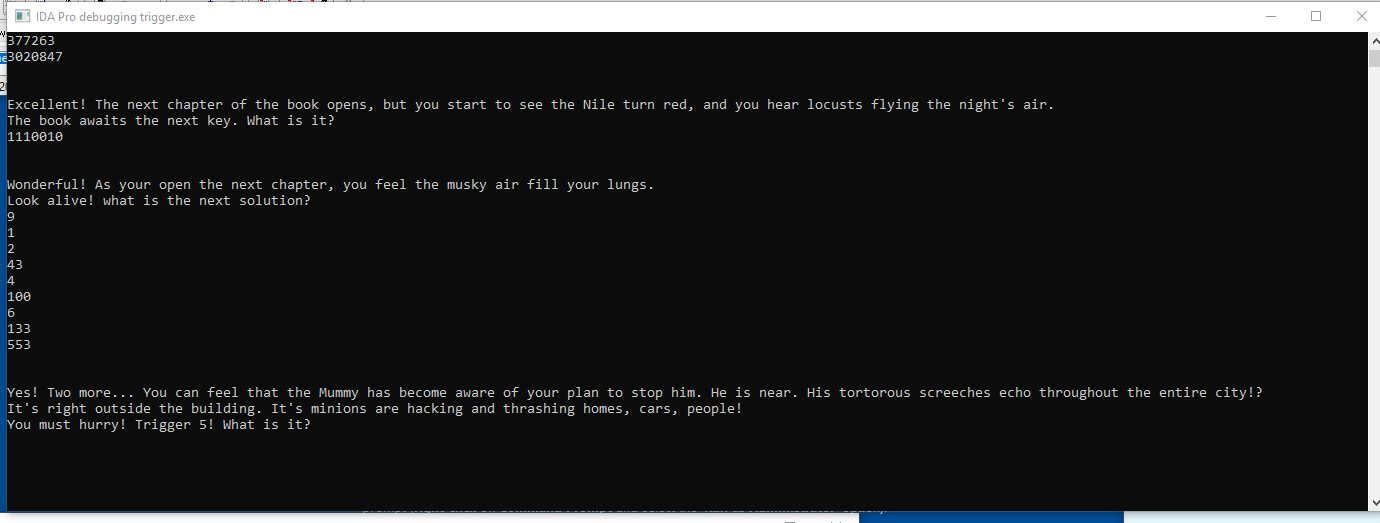
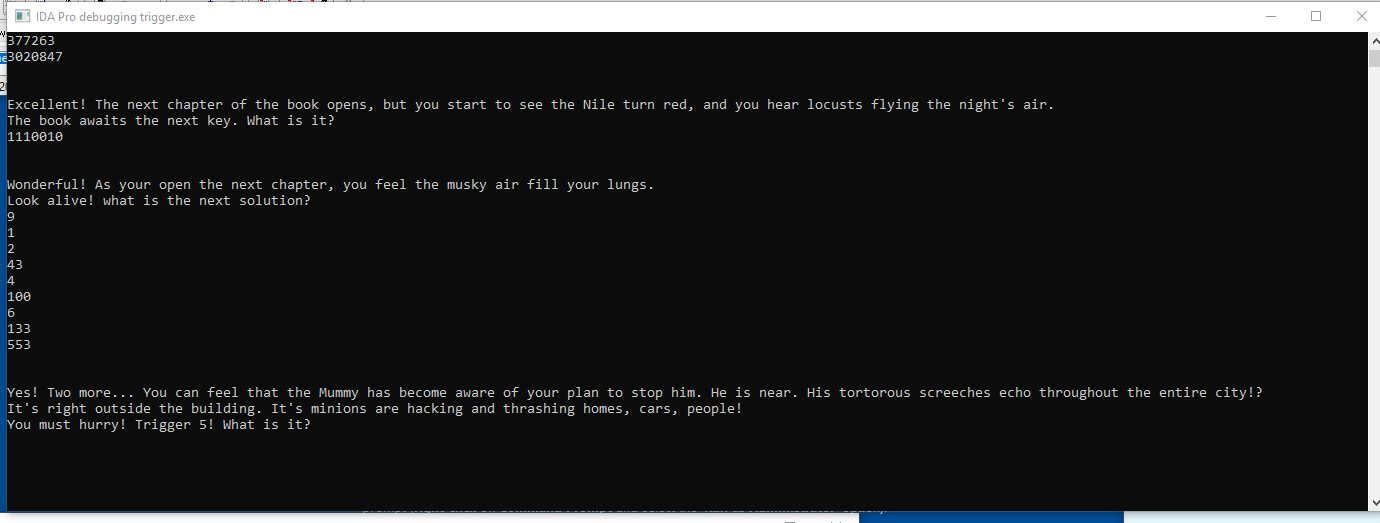
**When answering each question, be very explicit describing your steps and include any screenshots or other materials you think will help evaluate your work.** All files required for this assignment can be downloaded from the course web page: <http://cs5375.cs.utep.edu/>

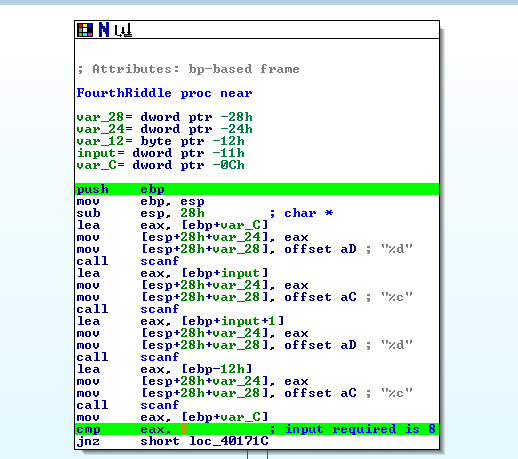
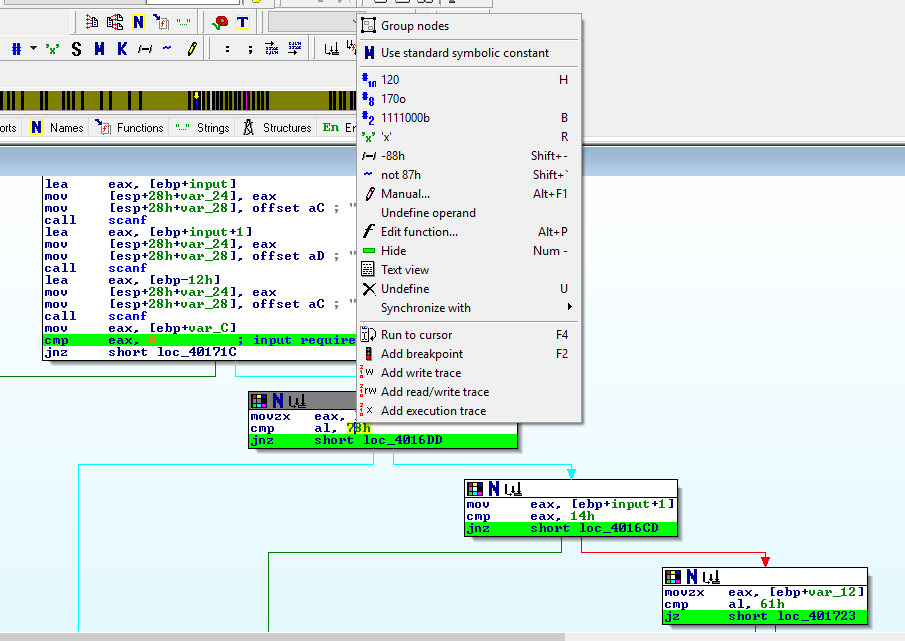
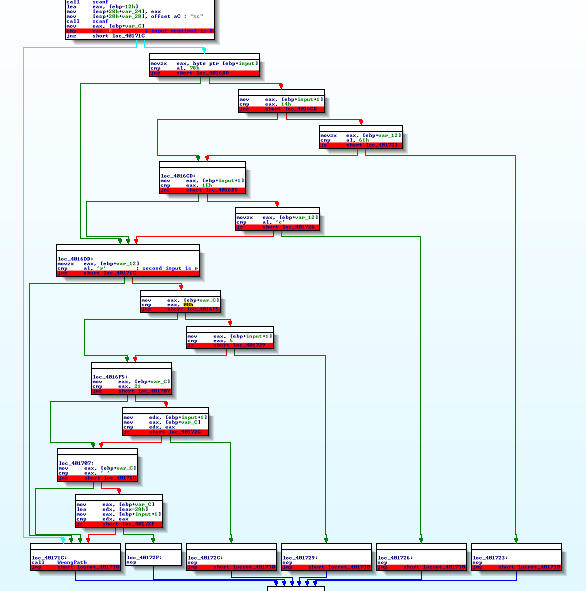
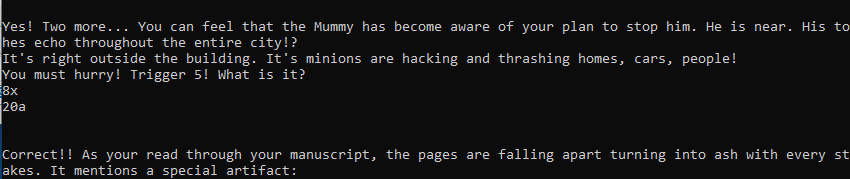
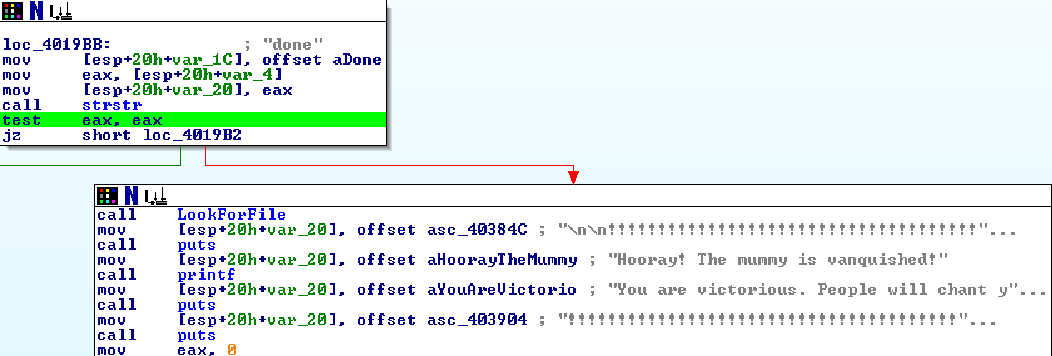
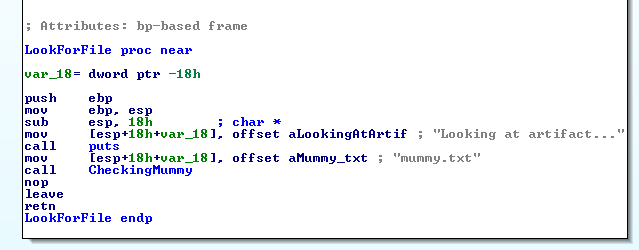
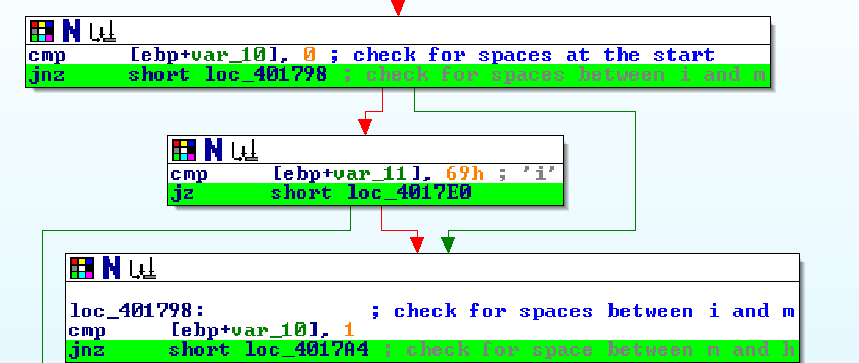
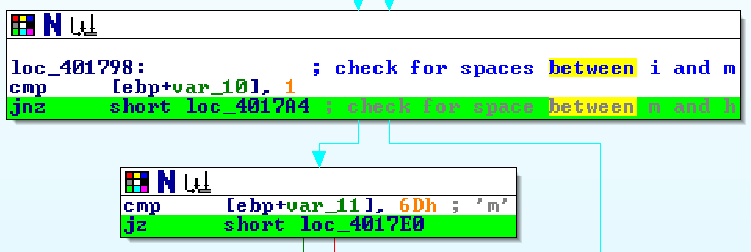
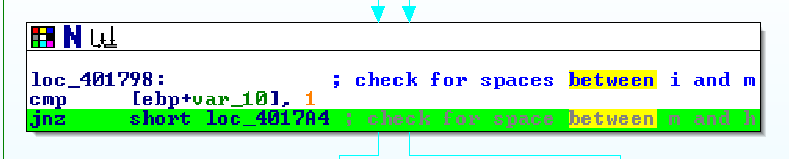
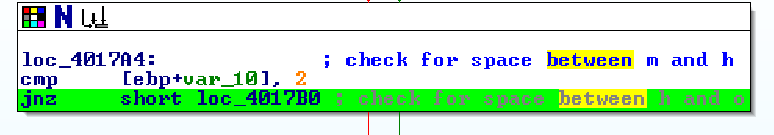
**Part 4: Assignment [Required]**

Answer the following questions:

1. What is the 1st secret passphrase? **-cs**
   * Find the main class and navigate to where its being called
   * 
   * 
   * Change your view into graph view using **space**
   * Look at how the structure of \_\_main looks like. It’s putting a lot of text into the console and doesn’t have any calls to subroutines until sub\_40128C
   * 
   * We then go into the subroutine, let us call it “firstProblem”. Looking at the graph view of the firstProblem we can all see that it goes into the same graph node.
   * 
   * IDA nicely gives us a comment, mostly likely a string. Try to input it into the console using “-cs!” sadly it fails. Then maybe look at the possible ways to avoid the Trigger Step 1?
   * Look for the location where the string “Trigger Step 1…” is being called inside the String Window already shown in your IDA Pro. Double click on the string sayings “!!!!!!!!Trigger Step 1…”. It will move you to where its being initialized. Look for the variable name, which is **aTriggerStep1So**, press X on it to see where it called and press OK. Rename the subroutine calling this to **TriggerStep**
   * 
   * Right clicking the values inside **loc\_4012AD** that aren’t 0 and 1 allows us to convert the hexadecimal values into ascii and do it with all possible options available.
   * 
   * Looking at getopt, we can assume there needs to be a parameter entered using command line. Try running the debugger with a random parameter (Going into Debugger, then process options allows you to add a parameter that IDA will give the console when running) and add a breakpoint at the first CMP you see within the subroutine named “firstProblem” and run the Debugger.
   * Hovering over var\_1C gives us what the value of our input is which 63h = c. Rename it into “input”.
   * 
   * Looking more into the file we can see ‘c’ ‘s’ and ‘!’ being checked as part of the parameter input. Execute each instruction (F8 to prevent going into getopt) to see the path of where the execution is going into. Going with -s fails, -c fails, -! fails. Could be a combination of it, so let’s try what the IDA comments which is -cs! Which also fails. Now we try with -cs seeing if that would make a difference.
   * Going with -cs passes the first secrete passphrase.
2. What is the 2nd secret passphrase? **7, 1, 2767, 11615, 58519, 377263, 3020847**
   * Going back into the \_\_main we can move past “firstProblem” as continue to the next subroutine call.
   * Find the subroutine being called before the “aExcellenetTheNe” and after “You have proved your worthiness…” (this is known due to passing the first secret passphrase) as the string seems to say that you passed the first riddle and rename the subroutine to “FirstRiddle”.
   * 
   * Going into the riddle we can see that will ask for an input using scanf and it’ll do a compare of EAX to the value 7 and if its not zero it’ll go into the **WrongPath** (named from Question 1)
   * 
   * We add a breakpoint at the cmp being done and run the debugger. It will ask for an input, we input 7, Go into the IDA View-EIP (pressing space puts it into graph view) to view the execution. From this point we are gonna get asked to input something again inside the scanf part.
   * 
   * The program will ask for 6 inputs, continue to use F8 to traverse through the program without going into the **scanf.** Now we will try adding something simple to continue to see the process. So, we try “123456”. After inputting the value 6 we are going to continue to traverse through the problem using F7 as we want to go into the endp of the “FirstRiddle”
   * 
   * Going into the next portion of debugging we can see **WrongPath** subroutine and want to avoid it. So, a comparison of ECX and EAX is not 0 then we know we messed up.
   * When you traverse through the execution you can see the values of ECX = 2 and EAX = 2676 or 0ACFh in hex when comparing ecx and eax (We also add a breakpoint here). The 2 seems to be from the value we input through the console.
   * 
   * 
   * So, we replace 2 with 2767 so when we compare it’ll equal 0 and not go into the wrong path. The input should be 7, 1, then 2767, The rest should continue to be 3,4,5,6 and then we traverse all over again.
   * By changing the 2 in our input to 2767 we can ignore the wrong path and it will traverse once more.
   * When you traverse through the execution you can see the values inside the comparison before the jump loc\_401389 (CheckInputs)the value ECX = 3 and EAX = 2D6FH in hex. The 3 seems to be from the value we input through the console.
   * 
   * So, we replace 3 with 2D5Fh or 11615 and the rest should continue to be 4,5,6 and then we traverse again.
   * Traversing again through the execution you can see the values inside the comparing ECX = 4 and EAX = E497.
   * 
   * We replace our input 4 with E497 and the rest should continue to be 5,6 and then we traverse again.
   * Traversing again through the execution you can see the values inside the comparing ECX = 5 and EAX = 377263.
   * We replace our input 5 with 377263 and the rest should continue to be 6 and then we traverse again.
   * Traversing again through the execution you can see the values inside the comparing ECX = 6 and EAX = 3,020,847.
   * And finally, we got the message of it being completed.
   * 
3. What is the 3rd secret passphrase? **1110010 or 10EFFA**
   * From the previous passphrase we get a hint of where to go next. Looking at the string windows we find the phrase that beings with “The book awaits the next key...”. It’ll show us the next variable we want to find its xref (Press X on the variable named **aTheBookAwatsT** and click OK). We find the next call being done and rename it to the “**SecondRiddle**”
   * 
   * We go inside the subroutine call. At first glance we can see it’ll ask for an input and it doing a comparison and jump to the correct path if the comparison equals to zero. We set a breakpoint at the comparison and run the debugger (When running the debugger make sure to input previous answers). We input a random number to see what the value needs to be inside the comparison to be 0.
   * 
   * We hover over the EDX = 5 and that seems to be our value while EAX = 10EFFA in hex or 1110010 in decimal seems to be the other value its comparing to.
   * We re-run the debugger and change our input or EDX to that of what's inside of EAX.

* What is the 4th secret passphrase? **9, 1, 2, 43, 4, 100, 6, 133, 553**
  + We first see **scanf**, which takes user input and saves it.
  + We then store this user input in EAX and compare it 9 with CMP
  + At this point we utilized the debugger and set break points at each compare and jump so that we could see exactly what the next segment to be executed was.
  + For our attempt to solve the riddle we put in random numbers and were able to determine 2 things
    - A for loop was executing
    - We could enter 8 different inputs
  + 
  + Once finished inputting the different values we then exit the loop and enter another function call which we named “**CheckingInputs**”
  + In **CheckingInputs** we found the numbers 43, 100, 133, and 553.
  + From this point we took a brute force approach and kept the 1-8 order while trying each number in the **CheckingInputs** at a different spot. Once we could confirm that one of those four numbers was in the correct spot we moved on to the next number and used the same approach until we got the correct final list.



* What is the 5th secret passphrase? **8x 20a**
  + We renamed this function to **FourthRiddle**. Here we noticed that there is a pattern of %d followed %c paired with a **scanf**.
  + 
  + Originally, we tried to input 2 different decimal numbers and 2 different characters however we got an automatic exit from the debugger.
  + The first CMP is comparing EAX to 8 so we knew that no matter what this input would be required
  + Since we believe the next input should be a character, we used the IDA conversion for 78h which is the ascii value for **‘x’**.
  + 
  + We entered this in as the second value however this did not result in the correct answer.
  + We looked at the comparisons being doing with AL and wondered why it stayed at 8 so when inputting another random number, it would change to that value. When then converting something like 8x we noticed that AL would change value but not in the same way when inputting 8. We then did 88 and out of frustration we tried putting a letter after it the value of AL changed into a number we didn’t understand.
  + We used <https://www.asciitohex.com/> to convert 8x into hexadecimal and ended up finding that same value that was inside the comparison of AL to 78.
  + The next thing we tried was pairing the two as **8x**
  + The debugger revealed that this input took us to the next block which was comparing EAX to the value **20**. We used the same process as we did for the 8x and found that **20a** was the correct second input. This revealed the next riddle.
* 
  + 
* What is the 6th secret passphrase? **i m h o t e p**
  + For the 6th secrete passphrase or the 5th riddle. We set a breakpoint until we saw the message of completing the secret passphrase
  + 
  + We noticed we needed to input done but after inputting that it till asked for an artifact.
  + So, we looked for the string “Looking at artificat...” and see where its being used.
  + We found the location, and IDA nicely gives us the name of the file it wants.
  + 
  + Has another call being done, rename it to CheckingMummy and going inside us gives another subroutine to look at. We press R on the hex values to see if they are able to change into ascii.
  + IDA was nice enough to tell us that those hex values where infact letters.
  + It checks for a string of letter which is **imhotep**
  + The other checks were trial and error on what they were doing as we guessed it could be **im hot eb**
  + We figured out that they were checking for spaces, but they were checking for spaces between letters and how many spaces being in between.
  + 
  + As this part would go in the direction, we wanted but still go into the wrong direction.
  + 
  + From here we figured out that the number specified in IDA where checking for the number of spaces and the value of it would indicate its space in between certain letters.
  + 
  + 
  + From here we assessed the phrase **i m h o t e p** and passed with the correct answers.

Complete the following:

* + Write java code to implement the 1st passphrase check in the same way as the assembly code.

* + Write java code to implement the 2nd passphrase check in the same way as the assembly code.

* + Write java code to implement the 3rd passphrase check in the same way as the assembly code.

* + Write java code to implement the 4th passphrase check in the same way as the assembly code.

* + Write java code to implement the 5th passphrase check in the same way as the assembly code.

* + Write java code to implement the 6th passphrase check in the same way as the assembly code.

**Deliverables**: Email a zip file with the subject **Assignment-DynamicAnalysis** to fall2020cs5375@gmail.com. The zip file must contain the following:

* + - For questions 1-6, a Microsoft Word document with written steps and screenshots detailing your thoughts and actions during your completion of each question.
    - For questions 1-6, your idb file (or. i64 if you’re using IDA Pro 7) that contains your comments, renamed functions, etc.
    - For question 7-12, a documented java file called Passphrase<#>.java, where <#> is the passphrase number.

**Important Note:** Your java code should make the checks for the solutions in the same way as the assembly; to the ***extent that it is possible with Java***. For example, do not simply implement a string compare, unless that is what the binary does. If there are multiple solutions, your code should also allow for multiple solutions. Lastly, your grade will factor in the readability of your code; use understandable variable names, good comments, and good coding style.

**Note**: No credit is given if you simply supply an answer, you must document your steps and your code in sufficient detail.