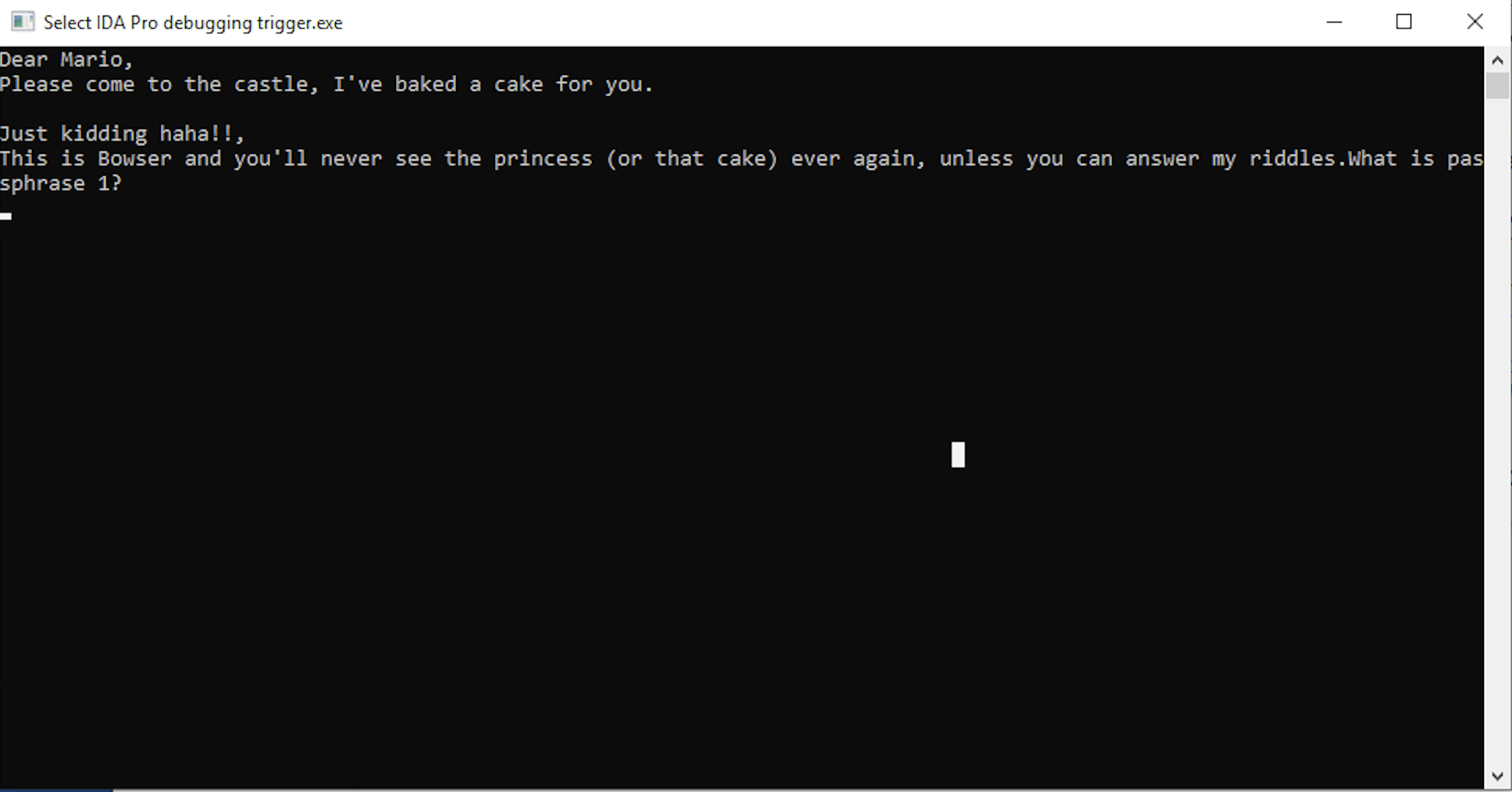
HW\_Dynamic

Due Oct 11, 2022

**Md Abu Sayed**, 80718658

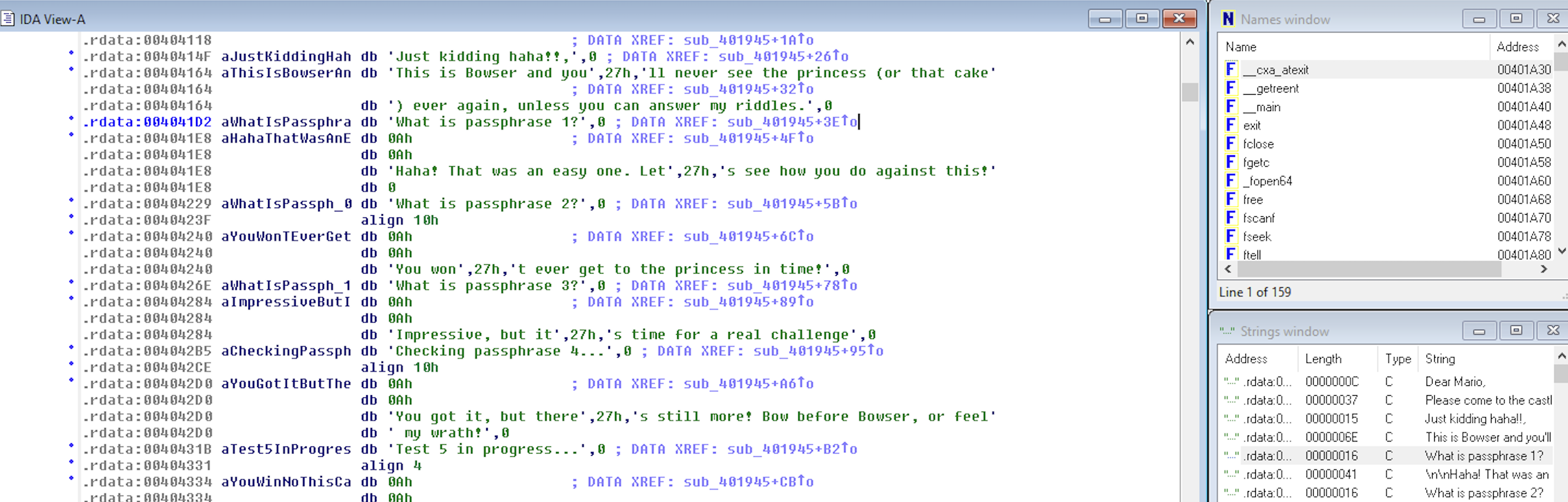
**Siyu Deng**, 80741923

# Q1.

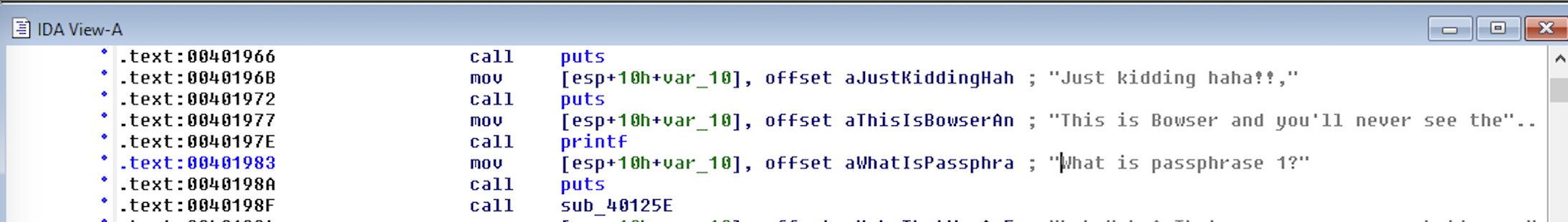
Use Debugger to run this program, and see the prompt question as following:

Randomly type in something, and see the program ends. It probably means we did not get the correct answer.

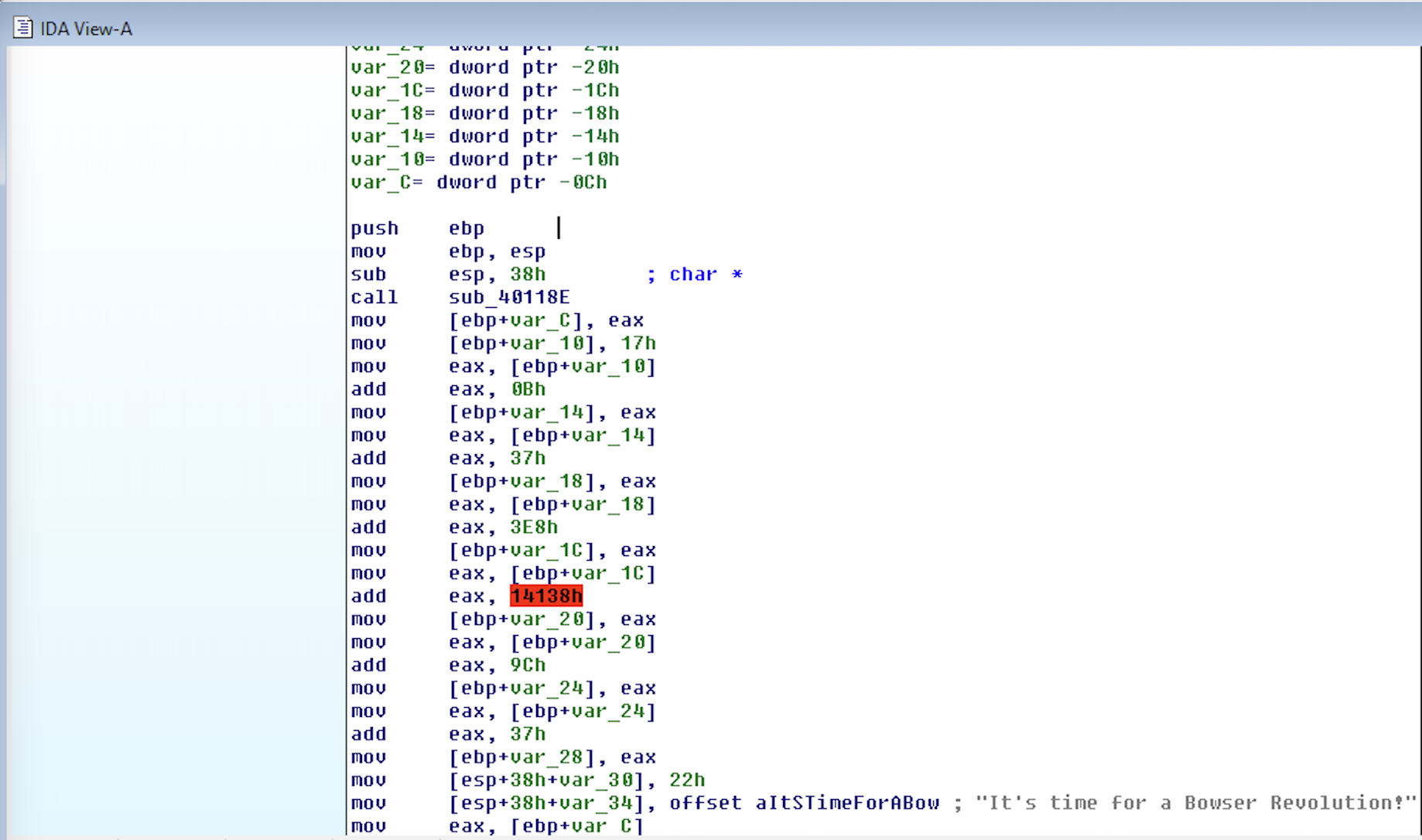
Use Strings window find the critical sentence “What is passphrase 1”. Double click it then we can see IDA pro shows this string has a cross-reference (eg. DATA XREF: sub\_xxxxxxx) which means it has been used by that function.



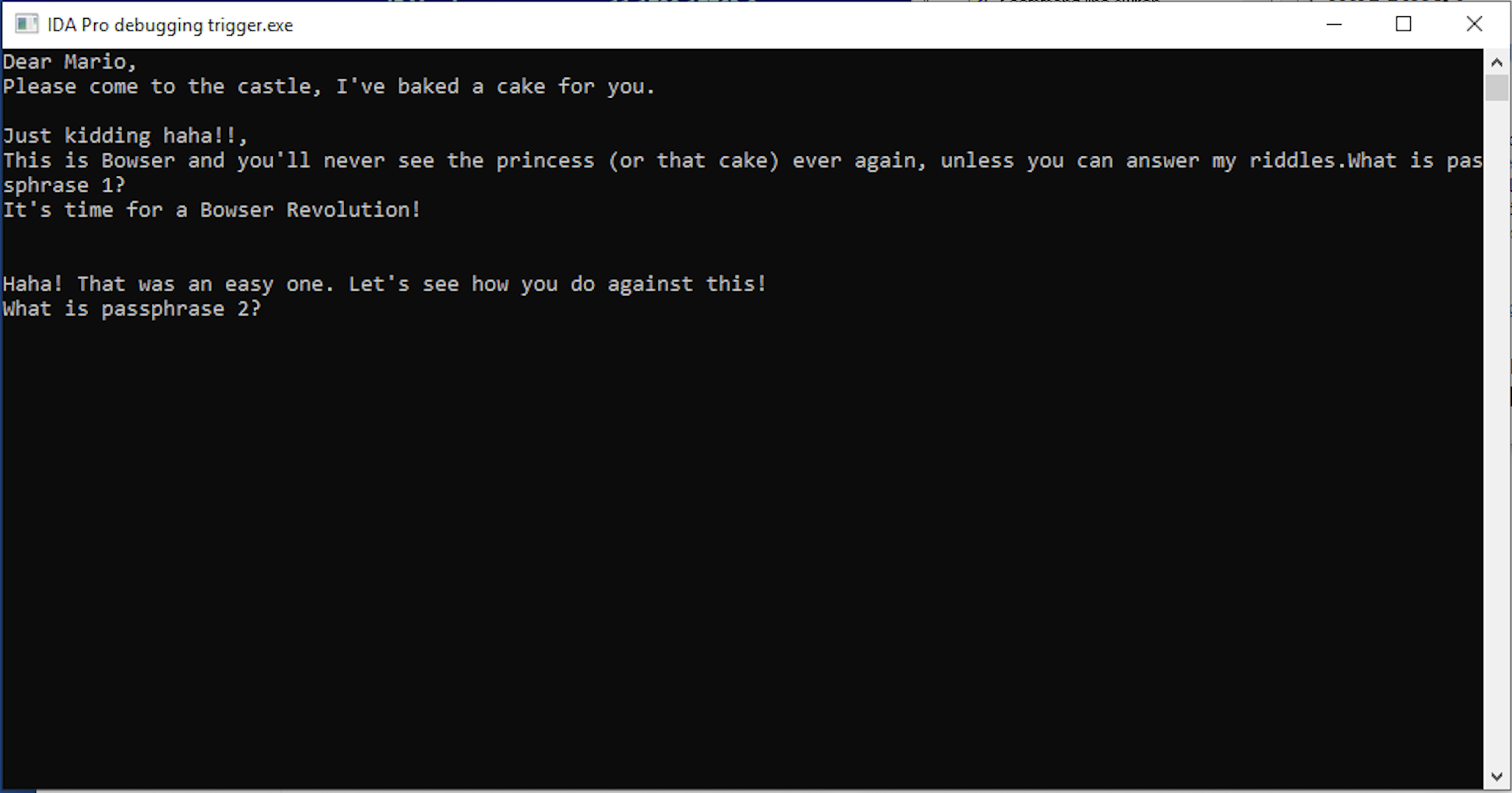
Double click the function name, then we are at the exact location where does this string has been used. By checking this function, we can find a function named “sub\_40125E” has been called to do the “passphrase checking” job. Now let’s click this function and go through what has it done.



By reviewing the execution code, we can see this function compared the input string with a string variable named “aItSTimeForABow”. And IDA Pro tells us the content of this string is “It’s time for a Bowser Revolution!”. This means such string is the answer for “passphrase 1”.



Now, let’s check our answer by running the Debugger again. After typed in this string, we see our answer is correct. And it asks for “passphrase 2” now.



Basically, the process we used to find out “passphrase” is like the following:

STEP 1: Identify which function asked for “passphrase X”

STEP 2: Identify which function checked the correctness of typed in “passphrase X”

STEP 3: Read through the execution code of previously mentioned function, and find out what is the correct answer for this passphrase.

I will skip STEP 1 & 2 for the rest 4 questions since these processes are identical with answering the first question.

# Q2.

Find out that function “sub\_401467” checks “passphrase 2”.

|  |  |  |
| --- | --- | --- |
| var\_28 = dword ptr -28h  var\_24 = dword ptr -24h  var\_12 = byte ptr -12h  … | * ptr: Pointer * dword ptr: fetch a doubleword (4 bytes) from the address * byte ptr: fetch a byte from the address | Declare required storage space for local variables |
| push ebp  mov ebp, esp | Frame preparation for current running function:   1. push the value in frame pointer ebp onto stack, and decrease (move up) the stack pointer esp address by the appropriate amount (in this case 4 bytes); 2. Copy address of stack pointer to the frame pointer. Now, the frame size is 0 and return address (previous ebp value) is stored below ebp (return address = ebp+4). | Function Prologue |
| push ebx | EBX is often used to hold the starting address of an array |  |
| sub esp, 24h | Allocate 24h bytes spaces for our function |  |
| lea eax, [ebp+var\_C/var\_10/-11h/-12h]  mov [esp+28h+var\_24], eax  mov [esp+28h+var\_28], offset aD/aC  call scanf | Scan values from user input and stored the scanned values into user defined addresses.   * aD means int %d * aC means string character %c   In this case:   * var\_28 is pointed by ESP which is the lowest address of this function frame. * var\_24 is used to store the scanned in value’s destination address * var\_12 and var\_11 are two 1 byte string character * var\_10 and var\_C are two 4 byte integer | Read user input |
| movzx eax [ebp+var\_11] | * movzx: similar to mov but move with zero extension which means if the src operand is smaller than the dest operand, the remaining bits in src are filled with zeros. Here value in var\_11 is 1 byte, and eax is a 4 byte register, so it uses movzx. | Load var\_11 value for the following preparation |
| cmp al, 62h | * al: is the least significant byte of eax, here means the value of var\_11 | Compare var\_11 with 62h (ASCII: b) |
| jnz loc\_40159F | - jnz: Jump if not zero | Jump to loc\_40159F if var\_11 is not b |

Now, let’s discuss if var\_11 = “b” first:

|  |  |  |
| --- | --- | --- |
| movzx eax, [ebp+var\_12]  cmp al, 64h | Similar to previous steps | Load var\_12 value and compare with 64h (ASCII: d) |
| jnz short loc\_40153F | * short jump means the jump destination is not far from current address (2 byte instruction, +/-127 away) * near jump means within segment jump | Jump to loc\_40153F if var\_12 is not d |

Now, let’s discuss if var\_11 = “b” and var\_12 = “d”:

|  |  |  |
| --- | --- | --- |
| mov eax, [ebp+var\_C]  cmp eax, 0Ch  jnz short loc\_401508 | Similar to previous steps | Jump to loc\_401508 if var\_C is not 12 |

Now, let’s discuss if var\_11 = “b”, var\_12 = “d” and var\_C = 12

|  |  |  |
| --- | --- | --- |
| mov ecx, [ebp+var\_C] |  | ecx = var\_C |
| mov edx, 66666667h |  | edx = 66666667h |
| mov eax, ecx |  | eax = var\_C = 0Ch |
| imul edx | * imul: signed multiplication of single operand   + - * imul edx = edx\*eax and stored in EDX:EAX which is a 64bit number (since both edx and eax are 4 bytes) | edx:eax = 4:CCCCCCD4 edx = 4h eax = CCCCCCD4h |
| mov eax, edx |  | eax = 4h |
| sar eax, 2 | * sar: shift arithmetic right * shr: shift logic right   It’s similar to divide by 2, in this case means eax = eax/4. | eax = 1h |
| mov ebx, ecx |  | ebx = 0Ch |
| sar ebx, 1Fh | - Since ebx is 4 bytes (32 bits), shift right by 31 bits means isolate leftmost bit, and since var\_C is unsigned integer, such bit is not representation of sign of var\_C. | ebx = 0h |
| sub eax, ebx |  | eax = 1h - 0h |
| mov edx, eax |  | edx = 1h |
| mov eax, edx |  | eax = 1h |
| shl eax, 2 | * shl: shift logic left * sal: shift arithmetic left   Similar to multiply by 2, in this case, multiply by 4. | eax = 1h \*4 |
| add eax, edx |  | eax = 4h + 1h |
| add eax, eax |  | eax = 5h + 5h |
| sub ecx, eax |  | ecx = 0Ch - 0Ah |
| mov edx, ecx |  | edx = 2h |
| mov eax, [ebp+var\_10] |  | Copy var\_10 to eax |
| cmp edx, eax |  | Compare 2h and var\_10 |
| jz loc\_4015E8 |  | Jump to loc\_4015E8 if var\_10 = 2h |

For this program, we find that functions loc\_4015F7, loc\_4015F4, loc\_4015F1, loc\_4015EE, loc\_4015EB, loc\_4015E8, loc\_4015FA will directly jump to the success function loc\_4015FB. And each of them has a single parent node. In other words, we will have 7 correct passphrase to pass this question.

Therefore, var\_11 = “b”, var\_12 = “d”, var\_C = 12 and var\_10 = 2 is one of the correct passphrase which should be written as “12 2bd”. And let’s write it into the terminal, it shows successfully passed.

For the rest 5 passphrase I will skip some of these line by line explanations, in summary:

| Gate | Path | Passphrase |
| --- | --- | --- |
| loc\_4015FA | var\_11!=“b", var\_11=“d”, var\_C!=2, var\_C=5, var\_12=“n”, var\_10=15 | 5 15dn |
| loc\_4015F7 | var\_11!=“b”, var\_11=“d”, var\_C=2, var\_12=“r”, var\_10=22 | 2 22dr |
| loc\_4015F4 | var\_11=“b”, var\_12!=“d”, var\_12 = “r”, var\_C!=11, var\_C!=21, var\_C=32, var\_10=-10 | 32 -10br |
| loc\_4015F1 | var\_11=“b”, var\_12!=“d”, var\_12 = “r”, var\_C!=11, var\_C=21, var\_10=var\_C=21 | 21 21br |
| loc\_4015EE | var\_11=“b”, var\_12!=“d”, var\_12 = “r”, var\_C = 11, var\_10=1 | 11 1br |
| loc\_4015EB | var\_11=“b”, var\_12=“d”, var\_C!=12, var\_C=234, var\_10=4 | 234 4bd |
| loc\_4015E8 | var\_11=“b”, var\_12=“d”, var\_C=12, var\_10=2 | 12 2bd |

Therefore, basically this function’s pseudocode looks like the following:

if var\_11 == “b”:

if var\_12 == “d”:

if var\_C == 12:

if var\_10 == 2:

return True

else:

return True

elif var\_C == 234:

if var\_10==4:

return True

else:

return False

else:

return False

elif var\_12 == “r”:

if var\_C == 11:

if var\_10 == 1:

return True

else:

return False

elif var\_C == 21:

if var\_10 == var\_C:

return True

else:

return False

elif var\_C == 32:

if var\_10 == -10:

return True

else:

return False

else:

return False

else:

return False

elif var\_11 == “d”:

if var\_C == 2:

if var\_12 == “r”:

if var\_10 == 22:

return True

elif var\_C == 5:

if var\_12 == “n”:

if var\_10 == 15:

return True

else:

return False

else:

return False

else:

return False

else:

return False

# Q3.

Find out that function “sub\_40134F” checks “passphrase 3”.

Similar to passphrase 2, this function declared 5 local variables, from bottom to up they are: var\_C, var\_10, var\_28, var\_34, var\_38. And all of them are double word which has size 4 bytes.

In sub\_40134F, var\_10 assigned with 6, var\_C assigned with 0.

Function loc\_401384 and loc\_401365 form a for-loop. As long as var\_C is less than var\_10, jump to loc\_401365.

In loc\_401365, the scanned integer is stored at location var\_C\*4 - 28h and increase var\_C by 1. Therefore, this loop will run 6 times, the scanned values (integer, offset aD) are stored at the following locations:

| var\_C | Order of scanned value | Stored Location |
| --- | --- | --- |
| 0 | 1st | -28h |
| 1 | 2ed | -24h |
| 2 | 3rd | -20h |
| 3 | 4th | -16h |
| 4 | 5th | -12h |
| 5 | 6th | -8h |

Then, the program will call function sub\_40118E. This function declared var\_C, var\_10, var\_14, var\_18, var\_1C, var\_20, var\_34, var\_38 and all of them are double word. We will go through some lines of this code to understand what does the function do for initialization.

|  |  |  |
| --- | --- | --- |
| … |  |  |
| mov [esp+38h+var\_38], 64h |  | esp = 64h |
| call malloc | * malloc function allocate space (here in data segment) with given size and return the space address to EAX. It will take the value stored on the top of the stack as input (size/bytes). * Usually used for making array | Allocate 64h (100) bytes space, or generate 100 bytes big array. |
| mov [ebp+var\_C], eax |  | var\_C = lowest address of array |
| mov eax, [ebp+var\_C] |  | eax = var\_C |
| mov [ebp+var\_10], eax |  | var\_10 = var\_C |
| mov [ebp+var\_14], 64h |  | var\_14 = 64h (100) |
| mov eax, [ebp+var\_14] |  | eax = 64h |
| mov [ebp+var\_18], eax |  | var\_18 = 64h |
| cmp [ebp+var\_C], 0  jnz short loc\_4011C6 |  | if allocated address is not 0, jump to loc\_4011C6 |
| var\_1C = dword ptr -1Ch |  |  |

Here is the table translated variables into human readable names:

|  | Renamed | Function |
| --- | --- | --- |
| var\_38 |  | Parameter space reserved for calling function |
| var\_34 |  | Parameter space reserved for calling function |
| var\_20 | NewArrayHead | Tracks new array Head after memory reallocation |
| var\_1C | NewByte | Hold newly read in byte |
| var\_18 | AvaliableSpace | Tracks available size remaining in array |
| var\_14 | ArraySize | Tracks array size |
| var\_10 | ArrayHead | Tracks current array Head |
| var\_C | ArrayPointer | Tracks array pointer which points to the next filling address |

Recall, the direction our array takes in values is from high to low.

Now, we are at loc\_4011C6. Functions loc\_4011C6, loc\_40121F, loc\_401235, loc\_40124F, loc\_401252, loc\_401253 are all together read byte by byte from a given file. In order to figure out what file does our program use, we put a breakpoint right before it calls fgetc function. And we find out that fgetc takes cygwin1.dll as input.

Recall, Dynamic Link Libraries (DLL)s are like EXEs but they are not directly executable. DLLs are MS's implementation of shared libraries. And DLLs are loaded into memory just once.

Here is the detailed explanation of what do they actually do:

|  |  |  |
| --- | --- | --- |
| loc\_4011C6 [1st part] | * fgetc() function reads a single byte from the given file. Returns –1 (0xFFFFFFFF is -1) on error. * \_\_getreent function is from newlib.c library. Re-entrant function can be interrupted in the course of execution and then resume later. Usually used in recursion. Here, \_\_getreent used to return reents (a struct variable) which is assigned to each thread. | 1. Read a byte from file stream (a sequence of bytes used to hold file data) 2. Check such byte is -1 or not. If it is -1, process to function loc\_40124F. Otherwise, process to rest lines |
| loc\_4011C6 [2ed part] | var\_18 tracks the available size of defined array | Check whether array has space to hold the incoming byte.   * If so, jump to loc\_401235 * Otherwise, process to rest lines. |
| loc\_4011C6 [3rd part] |  | Increase array size by 2 times of the original, and reallocate memory space.   * If the reallocation is failed, process to the rest lines. * Otherwise, jump to loc\_40121F |
| loc\_4011C6 [4th part] | - free: free previously allocated space | Free allocated space and jump to locret\_40125C. **[end]** |
| loc\_40121F |  | Update array pointer and array head after reallocation and process to loc\_401235. |
| loc\_401235 | - 0Ah: it’s the line feed indicates the end of line. | * If the newly read byte is line feed, jump to loc\_401252. * If not, jump back to the beginning of loc\_4011C6 |
| loc\_40124F | - nop: no operation | Jump to function loc\_401353 |
| loc\_401252 | - nop | Process to loc\_401253 |
| loc\_401253 |  | - Put 0 at the end of array.  - Copy ArrayHead to eax and process to locret\_40125C. **[end]** |

After the file read step is done, and we return to the main program. The main program loads var\_10 (6) and the user input variable var\_28 as parameters (var\_10 is at the higher position) to run function sub\_4012D8. Now, let’s look into what does sub\_4012D8 do.

At the initialization step, it declared 3 variables:

| Name | Address | Correspond value |
| --- | --- | --- |
| var\_C | -0Ch |  |
| arg\_0 | 8 | var\_28 = user input (an address) |
| arg\_4 | 0Ch | var\_10 = 6 |

Then it checks whether the value at user input is 78h (120), if so, it jumps to loc\_4012ED, otherwise it calls sub\_401170 which is the wrong message.

Now, start from loc\_4012ED, this program runs a for loop:

for(int var\_C=1; var\_C<var\_10; var\_C++)

At each iteration, it runs loc\_4012F6, let’s review what does it actually do step by step.

|  |  |  |
| --- | --- | --- |
| mov eax, [ebp+var\_C] |  | eax = var\_C = # of iteration = i |
| lea edx, ds:0[eax\*4] | * ds: data segment * 0[eax\*4] is the same as [0+eax\*4] | Get the address of the ith instance in data segment. We have:  edx = 4h \* i |
| mov eax, [ebp+arg\_0] |  | eax = 78h |
| add eax, edx |  | eax =78h + 4h\*i |
| mov ecx, [eax] |  | ecx = value(78h + 4h\*i) |
| mov eax, [ebp+var\_C] |  | eax = i |
| add eax, 3FFFFFFFh |  | eax = 3FFFFFFFh+i |
| lea edx, ds:0[eax\*4] |  | edx = 4h\*eax  Here, 3FFFFFFFh is a magic number, if it multiplies by 4 (shift left by 2 digts), the remaining number is (i-1)\*4h. Ex, when i = 1, edx = 0h. |
| mov eax, [ebp+arg\_0] |  | eax = 78h |
| add eax, edx |  | eax = 78h + (i-1)\*4h |
| mov eax, [eax] |  | eax = value(eax) |
| mov edx, 0CCCCCCCDh |  | edx = 0CCCCCCCDh |
| mul edx |  | edx:eax = 0CCCCCCCDh\* eax  Ex, when i=1, eax=78h. Then, eax\*edx = 6000000018. Therefore, edx=60h, eax=18. |
| shr edx, 2 |  | edx = edx/4  Ex, when i=1, edx=60h/4=18h |
| mov eax, [ebp+var\_C] |  | eax = i |
| sub eax, 1 |  | eax = i-1 |
| imul eax, edx | signed multiply | edx:eax = edx \* (i-1)  Ex. when i=1, eax=18h\*(1-1)=0h |
| shl eax, 3 |  | eax = eax\*8  Ex. when i=1, eax=0 |
| add eax, 23h |  | eax = eax + 23h |
| cmp ecx, eax  short loc\_40133F |  | Compare ecx and eax, if equal, increment i by 1, if not, wrong function is called. |

Based on the execution above, the way this program checks the passphrase is by comparing the following user input 5 integers sequentially (carried by ecx) with the first one (120) by using some specific rule. And such rule is based on the value loaded from DLL file Cygwin1.dll.

So, we can hack this passphrase integer after integer, at each iteration set a breakpoint right before “cmp ecx, eax” statement, and check the value carried by eax at that moment. Such value should be the correct answer.

Here are the comparing values at each iteration:

| Iteration | Compared value |
| --- | --- |
|  | 78h (120) |
| i=1 | 23h (35) |
| i=2 | 5Bh (91) |
| i=3 | 143h (323) |
| i=4 | 623h (1571) |
| i=5 | 2763h (10083) |

Therefore, the correct answer for Q3. is: 120 35 91 323 1571 10083

# Q4.

After successfully passed passphrase check, the program crashed immediately. We need to look into P4 main program first. Like before, we will skip the detailed explanation if it’s identical to the previous inferences process.

Here, the P4 is checked by running function sub\_401600. In the initialization block, it opens a file named “browsers\_junk.bin” with “r” (read-only) permission. Then, it stores the file pointer that identifies the file stream into var\_10.

And it checks whether this file exist, if not (var\_10=0), it will trigger the wrong passphrase function. This is the reason why our program terminated immediately right after passing passphrase3 check.

Before we create a fake “browsers\_junk.bin” to avoid the termination. Let’s check how does this function uses this file to pass the passphrase check.

At the second block of this function, it calls fseek, by looking up from the documentation:

fseek(FILE \*pointer, int offset, int position)

fseek() is used to move file pointer associated with a given file to a specific position. And there are three default positions:

* 0: [SEEK\_SET](https://devdocs.io/c/io), beginning of the file
* 1: [SEEK\_CUR](https://devdocs.io/c/io), current position
* 2: [SEEK\_END](https://devdocs.io/c/io), end of the file

In our case, this function is called as fseek(var\_10, 0, 2), which means the pointer has been moved to the end of the file with 0 offset.

Then it runs ftell function, according to documentation:

ftell() returns the current file position of the specified stream with respect to the starting of the file.

In our case, it’s called like ftell(var\_10) which will return the length of our loaded file. And then, it checks the length of file whether equals to 400h (1024), if not, the wrong passphrase function will be triggered.

Now, let’s process to the next block. By looking through this block, we can see it repeatedly read a byte from file stream at specific positions. It stored these data at var\_11, var\_12, var\_13, var\_14, var\_15. Any failure comparison will trigger the wrong passphrase function.

The following table shows the compared values with the corresponding pointer positions:

| Variable | Value | Position |
| --- | --- | --- |
| var\_11 | 70h (p) | 46h (70) |
| var\_12 | 65h (e) | 8ch (140) |
| var\_13 | 61h (a) | 0AAh (170) |
| var\_14 | 63h (c) | 0A8h (168) |
| var\_15 | 68h (h) | 3E7h (999) |

Therefore, in order to pass this question, we need to generate a file named “browsers\_junk.bin” and put it in the same directory as the .exe file. We used python code to generate such file:

import random

import string

# randomly generate 1024 length array with character entry

s = random.choice(string.ascii\_letters) for \_ in range(1024)

# replace letters at desired positions

s[70] = ‘p'

s[140] = ‘e'

s[170] = 'a'

s[168] = 'c'

s[999] = ‘h'

# convert to single string

s = ‘'.join(s)

# write to file

f = open("browsers\_junk.bin", “w+")

f.write(s)

f.close()

Then, we successfully passed the check.

# Q5.

Now, let’s check the function for passphrase 5. We can see it sequentially run two functions sub\_40118E and sub\_401808. We have went though sub\_40118E for passphrase hacking. It loads cygwin1.dll file. Let’s check what does sub\_401808 do then.

In sub\_401808, it generates some string and use it as the input parameter to run function sub\_401759. Let’s check what is that string. The variables are initialized as the following:

| Variable | Value |
| --- | --- |
| var\_C | “bios” |
| var\_10 | “get serialnumber” |
| var\_38 | 63696D77h dword (4bytes) |
| [var\_38+4] | 20h word (2bytes) “[space]” |

Then, it called strcat(var\_38, var\_C) which appends var\_C to var\_38 to update var\_38. And then called strcat(var\_38, var\_10) which appends var\_10 to var\_38 to update var\_38.

Then, it called function sub\_401759. This function create a file named “text.txt” to write. Then it called fwrite(“mama mia”, 1, 8, file \*stream) which means sequentially write each character in the string buffer to file “text.txt”. The number of objects to be written is 8 (include [space]), and size of each object is 1. Then call fclose to close the file and return to sub\_401808.

Then, in sub\_401808, it runs sub\_40118E again to load the dll file. Following from that, it calls function sub\_4017A7.

In sub\_4017A7, it is initialized by open the “text.txt” file. var\_C holds the file stream. It calls fscanf(FILE \*stream, “%s”, addr(var\_10B)) which means reads a sequence of non-whitespace characters and stored in var\_10B.

Then, it checks the position where string “toadstinks” first occurs in var\_10B by calling strstr(var\_10B, “toadstinks”). And the position is returned to eax. If such string is found occurred in the file, then the passphrase check passed.

The trick here is we need to pause the execution after “text.txt” file has been generated. And add string “toadstinks” in the middle of “mama”. For example, one acceptable updated string in “text.txt” file looks like “matoadstinksma mia”. Then, we resume the debugger, we can see the passphrase check passed.

