CS 3844 Computer Organization - Lab #05 Name/abc123:\_\_\_\_\_\_\_\_\_\_

This lab continues introducing you to the x86 Intel instruction set. Set up a new project in Visual Studio using the Lab5.cpp file and the Secret.h header file – or just reuse Lab #4. Create an Empty Windows Console application and “Add Existing” to make Lab4.cpp part of the project under Source files, and the Secret.h under Header Files.

Make sure to review PUSH/POP/CALL/RET instructions prior to doing this lab.

Also we can disable address space randomization byte going to project properties and disabling it. This *should* make everyone’s EIP the same – but as we saw in Lab #4 – it didn’t. That’s OK.

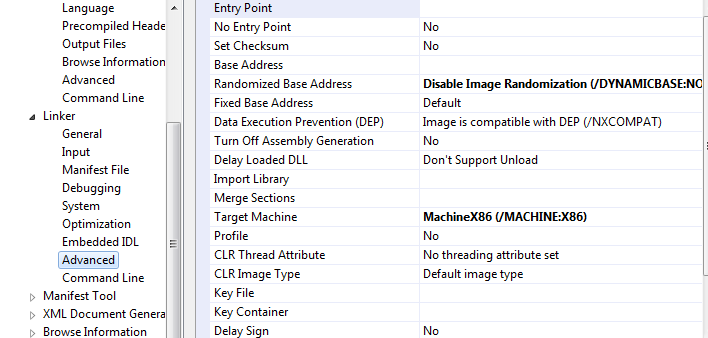


Figure 1 - Disable ASLR

This is all the same from Lab#4.

*Once you have the project compiled, go ahead and run it and observe what it does. This program requires one argument, a search string. There is a “secret” string in the header file. This program will take the input string, search the secret string, and return true if found, false otherwise. It will remove any part of a word. So in the word “Roadrunner” it would find the strings “Road”, “run”, “runner”, and even “ad.” It IS case sensitive. IT will find any sequence of characters, so it would also find “oad” for instance.*

*To play, run without any arguments and it will print some hints. So run it by guessing a string and seeing if your string is in the secret sentence. It does not have to be a valid word. Get enough guesses right and you can guess the secret sentence. Get too many wrong, and you will get frustrated and peek at the .h file – I know you will!*

*To run it within the Visual Studio debugger (required to set the breakpoints) you need to tell Visual Studio that you have an argument. Go to the property pages as shown in Figure 1 and enter the argument “computer” which is part of the secret sentence.*

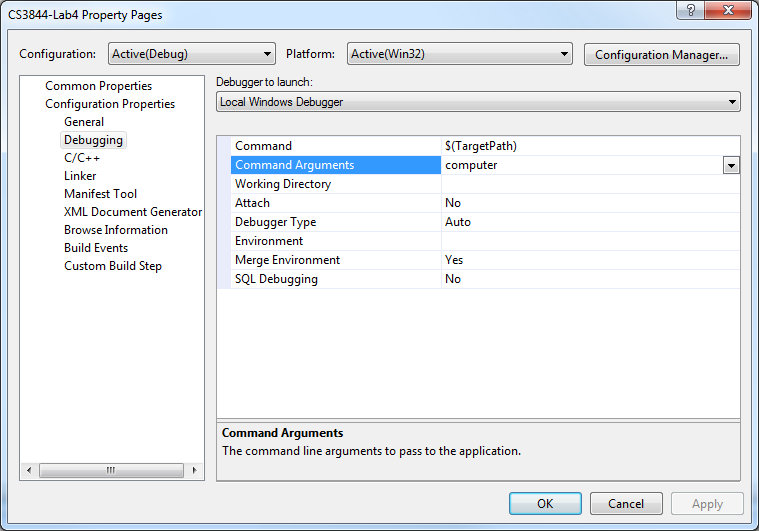


Figure 2 - Set Up Command Argument

Within Visual Studio, now run the program and see what it does. Next, set a breakpoint on the call to searchString as shown here and run the program in the debugger. Review Lab #4 with the solutions and make sure you comprehend what was done and why. If not, ask questions. When the exam comes, I’ll be asking the questions. Trust me, YOU are not the only one with these questions!

Now, let’s move on to Lab #5. Go to the searchString Function and set a BP on “bool result = false.” Step into searchString and select the Disassembly tab. It should look something like Figure 3 – Search String Code.

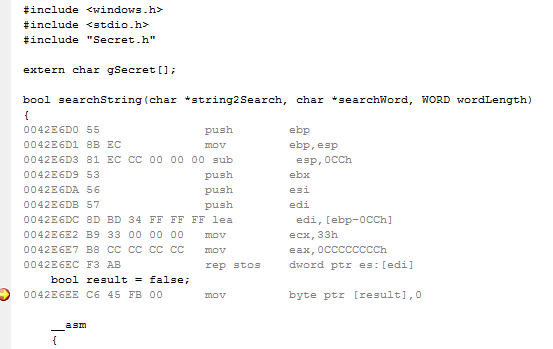


Figure 3 – Search String Code

Display ESP in a memory window which should look similar to Figure 4.

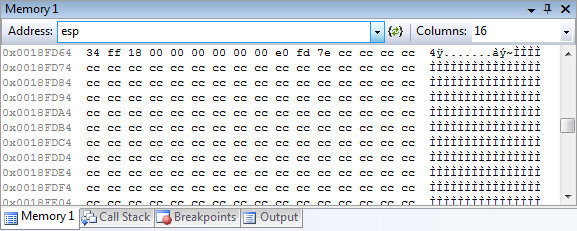


Figure 4 - Stack for Search String

Notice all the 0xCC values? It is often an exam question as to what they are and why they are there – it’s not magic nor is it random. It is *briefly* discussed in the slides and narrations: F. Inline Program Examples and narration F2. Inline Program Examples. HOWEVER, that is not sufficient so I made a new narration to go with this lab for that purpose. D1. x86 Intel Architecture - Detailed Stack - DEMO (Prelude for Lab #5). If you have not watched this exciting 30 min video, it’s *highly* recommended before doing Lab #5.

For Lab #5, we are going to ignore the security code. We set up the stack frame with push ebp, mov ebp,esp, and sub esp,0xCC. This saves the old ebp, cements the epb stack frame for this function, and reserves space for our one local variable, “result.”

The disassembly for result doesn’t show the details of where result is on the stack. I can tell from the machine code where it is (you will not be required to do that) and I can tell you it is at ebp -5. (Machine Code: C6 45 FB 00 mov byte ptr [result], 0. I know the 0xFB is the displacement for ebp and the zero is the immediate value of zero. Hence this should be: mov byte ptr [ebp+0xFB], 0. 0xFB is a -5. So let’s ee if that is true. In a memory window do an ebp-8 and then do a single step to complete the mov instruction. Did you see the CC value turn to a red 00? Note: ebp-5 is a larger number and therefore to the right of ebp-8.

Now you should be at the hand-written assembly.

1. What is the value of eax after executing the xor instruction?
2. Step down to the je NOT\_FOUND instruction (jmp if equal to zero). Google the test and je instructions and see what they do. Then apply that to test esi,esi, je NOT\_FOUND.
   1. What is the value of searchWord? (may vary from machine to machine)
   2. What is the offset from ebp of searchWord? (This will ALWAYS be the same. The offset is in the machine code, BUT from the video, you can tell what it is without looking at the machine code.)
   3. What value do you expect the ZERO flag to be after the test instruction? Why?
   4. Will the je be taken? Why or why not? (“taken” means it will jump to the code at NOT\_FOUND rather than execute the next instruction)
   5. In a memory window, type in &searchWord.
      1. What is the address of searchWord? (may vary)
      2. What is the value at that address? (may vary)
      3. Subtract ebp from the address of searchWord. The result will be the same for everyone and should equal your answer to #b. above. What is the result?
   6. The value of searchWord (#ii. above) is an address, so type “searchWord” in a memory window. What is in memory at that location?
3. Single step to the next je instruction. It is a repeat of the prior set of instructions except this one is checking string2Search rather than searchWord.
   1. What is the value of the offset from EBP for string2Search? (Again, it’s in the machine code but you should know based on the function call.)
   2. Based on the test instruction and the value of edi, will the jump be taken?
   3. In a memory window, type in string2Search. What is in memory at that location?
   4. Make a high level comment to describe what this series of instructions does. Consider that the function take two addresses as arguments. This functionality can be described in on brief sentence.
4. Neither of the jumps (je == jump if equal) are taken. Single step to the move instruction. Now figure out what you need to do to answer these questions:
   1. What value do you expect to be in al after the move?
   2. What is the test al,al instruction doing? Consider that searchWord/searchString are C style strings.
5. Single-step to the first instruction in MAIN\_LOOP:.
   1. What do you expect al to equal after the “**mov al,byte ptr [edi]**” instruction?
   2. Based on the value of al and the test al,al instruction, will the je be taken?
6. Single-step to the “**cmp al,bl**” instruction.
   1. What are the values of al and bl?
   2. The cmp instruction will do this: al – bl and set flags. The difference is not stored. (If you want to store the difference, use sub al, bl.) Based on the values of al and bl, which flags will be set/cleared?
   3. Will the jne NEXT\_CHAR be taken?
7. Set a BP on the “**call CHECK\_THE\_WORD**” instruction. Press “continue” (the little green debugging arrow) and execution will stop at that call.
   1. How many parameters are pushed?
   2. What is the return address that will be pushed to the stack once the call is made?
   3. Step into the call. Show the top of the stack as it appears in memory?
8. Comment the code. You don’t need to comment each line, but you can group in blocks since several instructions may perform a single task. I’ve grouped it for you this time.

CHECK\_THE\_WORD:

0042E72B 57 push edi

0042E72C 56 push esi

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

0042E72D 66 8B 4D 10 mov cx,word ptr [wordLength]

0042E731 0F B7 C9 movzx ecx,cx

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

0042E734 B0 01 mov al,1

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

0042E736 49 dec ecx

0042E737 74 13 je EXIT\_CHECK\_THE\_WORD (42E74Ch)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Under what condition will the above “je” be taken? This ties into why we set al == 1 (true)

WORD\_CHECK:

0042E739 46 inc esi

0042E73A 47 inc edi

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

0042E73B 8A 07 mov al,byte ptr [edi]

0042E73D 8A 1E mov bl,byte ptr [esi]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

0042E73F 3A C3 cmp al,bl

0042E741 75 07 jne NO\_MATCH (42E74Ah)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

0042E743 49 dec ecx

0042E744 75 F3 jne WORD\_CHECK (42E739h)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

0042E746 B0 01 mov al,1

0042E748 EB 02 jmp EXIT\_CHECK\_THE\_WORD (42E74Ch)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

NO\_MATCH:

0042E74A 32 C0 xor al,al

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

EXIT\_CHECK\_THE\_WORD:

0042E74C 5E pop esi

0042E74D 5F pop edi

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

0042E74E C3 ret

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Put a BP on the “return result” right after the label “DONE:” Continue until you hit that BP. What is the return value? (i.e. the value of “result”) NOTE: this is NOT the return address. The return value is, by convention, in al, ax, or eax depending on the size of the return value.
2. Figure out where you can put a break point to see the return address before the function returns. What is that return address? (may vary)