## CS 3844 Computer Organization - Lab #04 Name/abc123: Ariel Guevara @aes804

This lab continues introducing you to the x86 Intel instruction set. Set up a new project in Visual Studio using the Lab4.cpp file and the Secret.h header file. (If you want to play, don't peek at the Secret.h file.) 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.

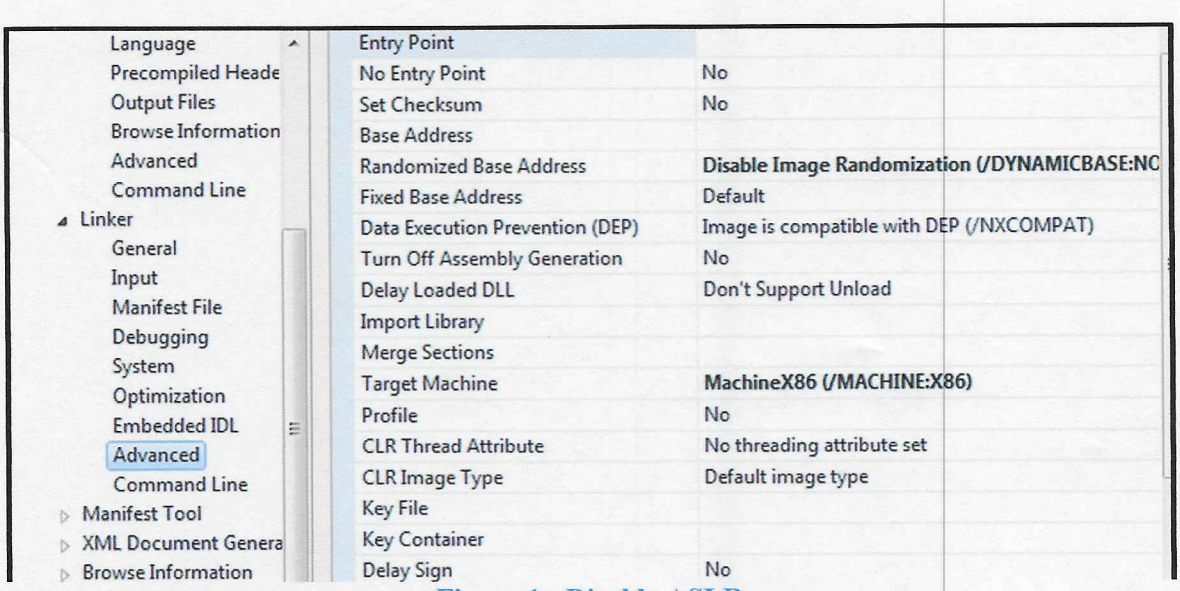



**Figure 1 - Disable ASLR**

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.

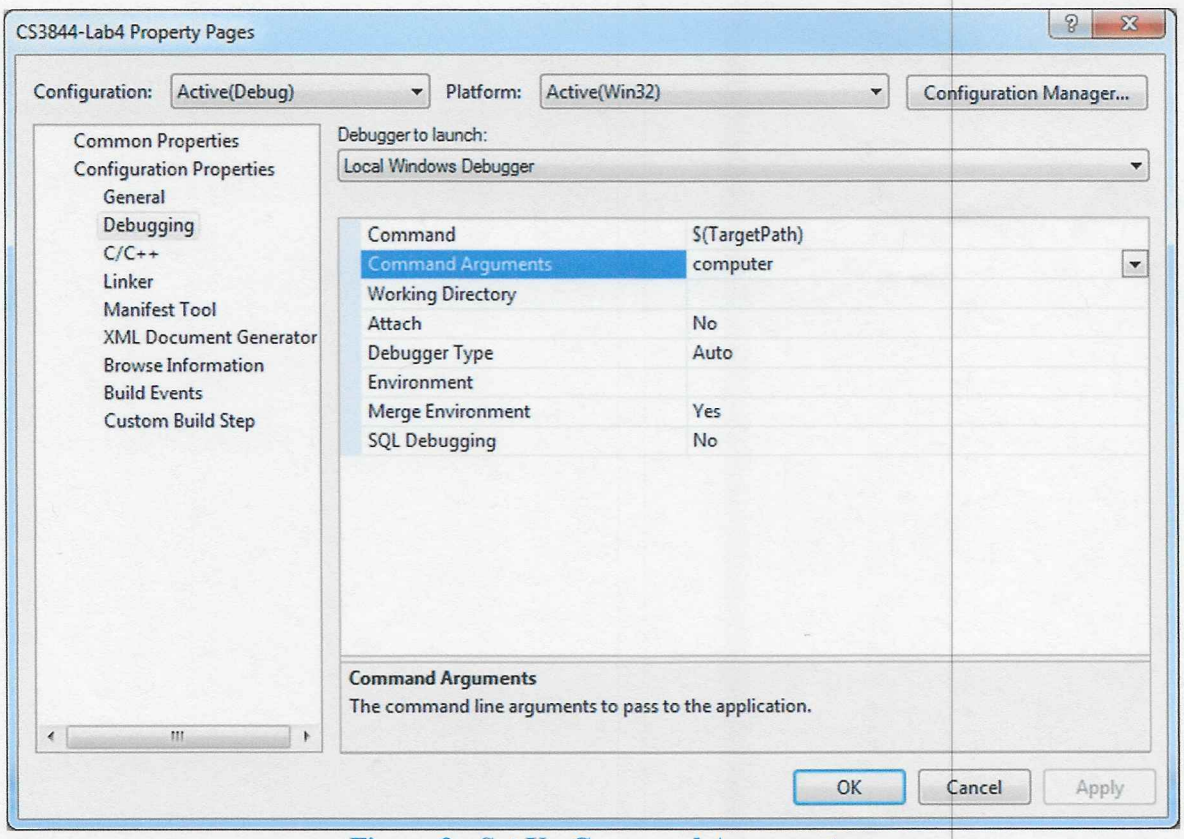



**Figure 2 - Set Up Command Argument**

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.

Disassembly | Lab4.cpp | Secret.h

(Global Scope)

```
int main(int argc, char *argv[])
{
    bool stringCompareResult;
    int wordLength;

    if(argc != 2)
    {
        printf("\n\nEnter the program name followed by a string to help you guess the secret string.\n\n");
        printf("Hint: Computer vs. Human in a game, who won?\n");
        printf("Hint: Mike Tyson player soccer.\n");
        printf("%s <your string, no spaces>\n\n", argv[0]);
        system("pause");
        exit(-1);
    }

    wordLength = strlen(argv[1]);

    stringCompareResult = searchString(gSecret, argv[1], wordLength);   // this length is arbitrary

    if(stringCompareResult)
        printf("Success! The string \"%s\" is in the string!\n\n", argv[1]);
    else
        printf("Failure - your string \"%s\" was not found.\n\n", argv[1]);

    system("pause");
    return 0;
} // main
```

**Figure 3 - C Source Code**

See the Disassembly tab in the top left. If not, go to Debug\windows\Disassembly (Alt-8). Click on the Disassembly tab.

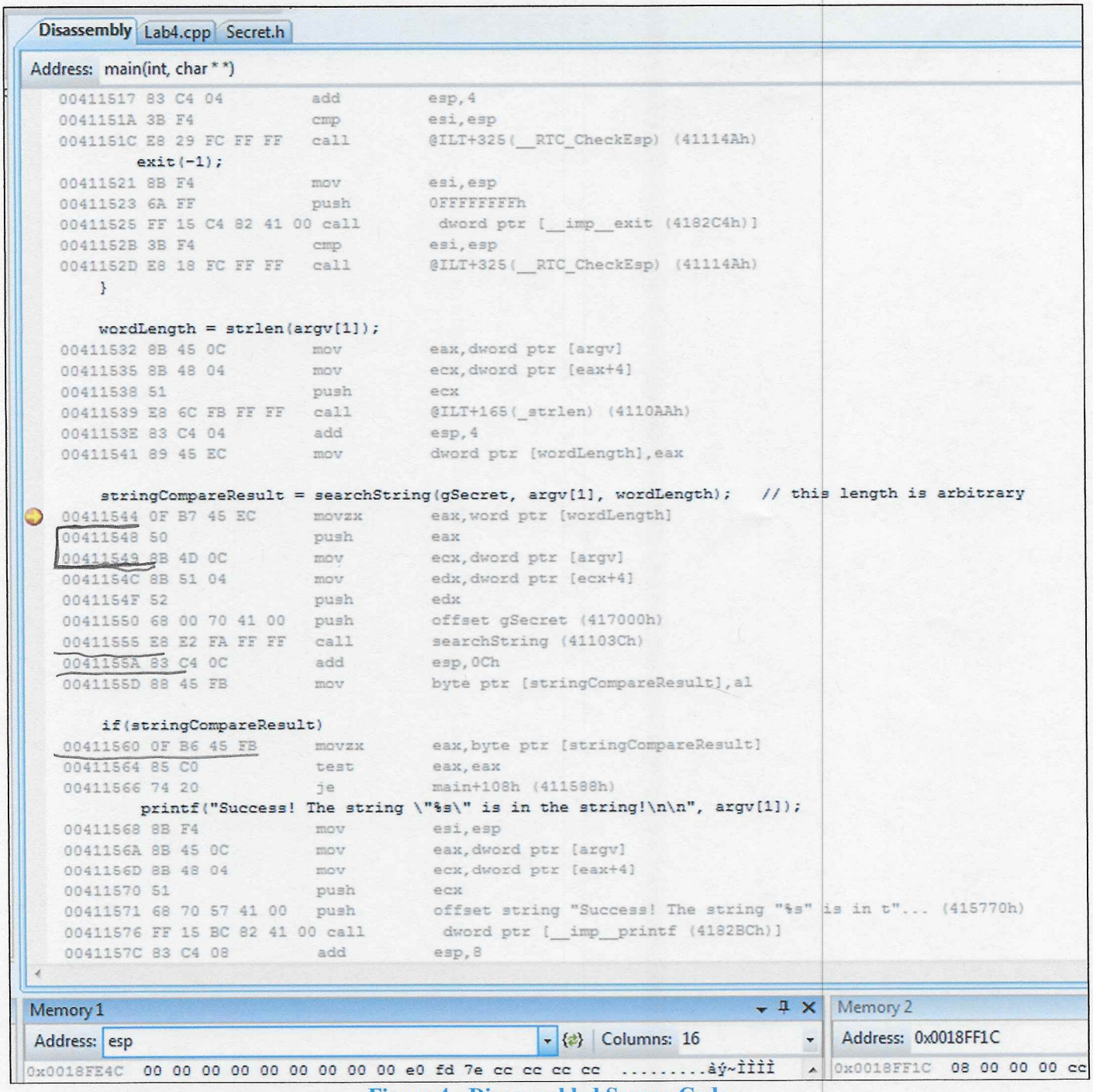


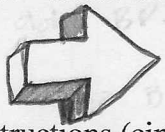

**Figure 4 - Disassembled Source Code**

1. Notice that I put ESP in the memory window. From Figure 4 what are EIP and ESP? Your EIP should match if you disabled ASLR but your ESP could vary, which is OK.

   EIP = 00434D65

   ESP = 0018FE50

2. Set a breakpoint on the "call" instruction (eip = 0x411555), on the add instruction following the call (eip=0x51155A), and on the movzx instruction after the "if" statement (eip=0x411560). (Note, due to a compiler inefficiency, the prior instruction already moved eax into "wordLength" and now it's moving "wordLength" back into eax, so even though this mov instruction has not been executed, eax already has the value of "wordLength.") What is the value of eax? Where does this number come from?

   EAX = 00000008

   length of 'computer'

3. Take two steps to execute the move and push instructions (eip = 0x411549). Type esp in the memory window again. What is the new value of esp and show the 4 bytes of memory at esp. AND, what is the difference between the value of esp now and prior to the push.

   ESP: 0018FE50
   mem: 00000000
   [prior]

   ESP: 0018FE4C
   mem: 00000000
   [now]

   50 10
   - 4C
   4
   [diff]

   4

4. You can single-step or run to the breakpoint so that the yellow arrow is on the call instruction. You see that we have two more pushes. Note: The number of pushes corresponds to the number of arguments and the arguments are pushed in reverse order. (ALERT ALERT ALERT!!! The fact that the C compiler standard is to push arguments in a reverse order is a major concept and often appears on exams.) We will discuss how "argv[ ]" works at a later time, accept for now it is pushing the address of the command line argument.

   a. In the memory window look at esp. Show the stack address (value of esp) below and twelve bytes that follow it.

      ESP = 0018FE44

      word length | 'computer' | gSecret

      mem: 00 00 00 08 00 25 FE F2 00 4A E0 00

   b. Note that at esp + 8, the word length value is there. Two other values have been saved to the stack. The first one saved is in edx and the other is a global variable address saved directly. You can put edx in the memory window. What are the addresses of "wordLength" and "gSecret?" HINT: the addresses should NOT be shown in little endian. Little endian is ONLY applicable when showing the contents of memory (like above, they are in memory and so are in little endian format.)

      EDX = 0054FEB0

      gSecret: 004AE000

      word length: 00000008

   c. When the CPU returns from the subroutine, it needs to know where to go. The return address is always the address immediately following the call instruction. What is the return address for this call instruction?

      return address → 00434D83

      (next instruction, which is add)

   d. Step into the call. It takes you to a jmp, so take another step. You should see something like in Figure 5. It should be on a push EBP instruction. We will go into this subroutine in Lab #5. For now, what is the new value of esp and its 16 bytes of contents.

      ESP = 0018FE40

      W. len. | 'computer' | gSecret | return addr.

      mem: 00 00 00 08 00 25 FE F2 00 4A E0 00 00 43 4D 83

      add esp, 0Eh


**Figure 5 - searchString Subroutine**

e. Given that the return address is at the top of the stack, what is the return address? Yes, it should match your answer above.

→ 00 43 40 83

5. Click on run to take you back outside the call. You should see something like in Figure 6. What is esp at this point? What is the purpose of this add instruction and why is it adding 0x0C? Type "esp-4" in the memory window, is the return address still there?

It pops off the 3 things we pushed before the return address (word len, address of gSecret)

and yes it is still there (we didn't mess with anything)

```
    stringCompareResult = searchString(gSecret, argv[1], wordLength);   // this length is arbitrary
00411544 0F B7 45 EC      movzx       eax,word ptr [wordLength]
00411548 50               push        eax
00411549 8B 4D 0C         mov         ecx,dword ptr [argv]
0041154C 8B 51 04         mov         edx,dword ptr [ecx+4]
0041154F 52               push        edx
00411550 68 00 70 41 00   push        offset gSecret (417000h)
00411555 E8 E2 FA FF FF   call        searchString (41103Ch)
0041155A 83 C4 0C         add         esp,0Ch
0041155D 88 45 FB         mov         byte ptr [stringCompareResult],al

    if(stringCompareResult)
00411560 0F B6 45 FB      movzx       eax,byte ptr [stringCompareResult]
```

**Figure 6 - Call to Search String Function**

6. What is the value in al?

al = 01

7. Type "&stringCompareResult" in the memory window. What are the contents of memory?

01 cc cc cc

8. Take two more steps to reach the last breakpoint (eip=0x411560). Type "&stringCompareResult" in the memory window. What are the contents of memory now?

cc cc cc 01

9. Lab 5 will explore the subroutine and what happens inside the call.