

@aes604

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 by going to project properties and disabling it. This should make everyone's EIP the same.

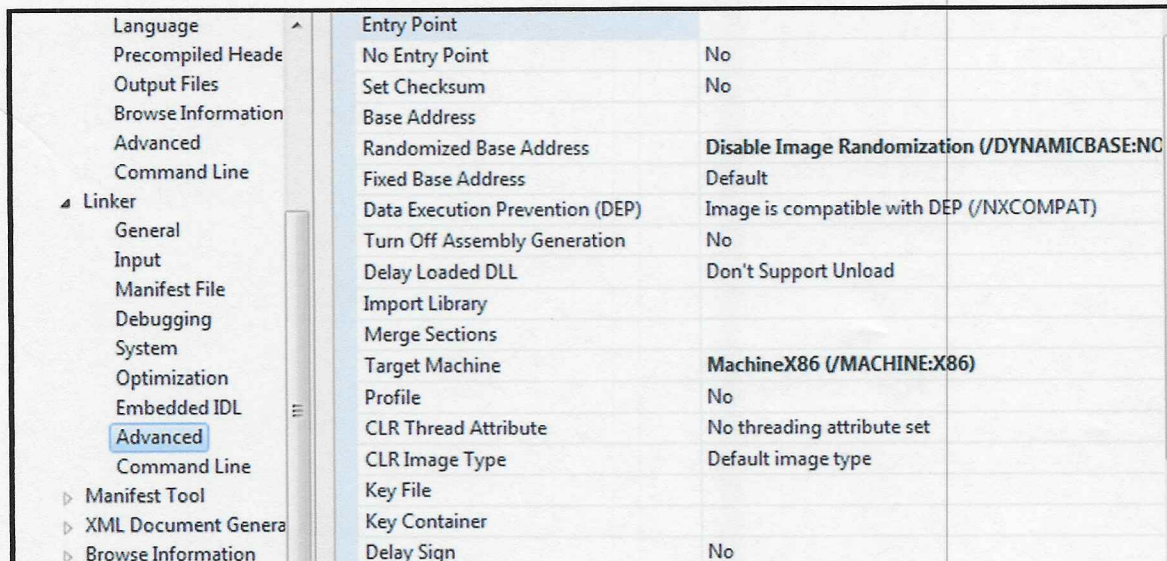


Figure 1 - Disable ASLR

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.



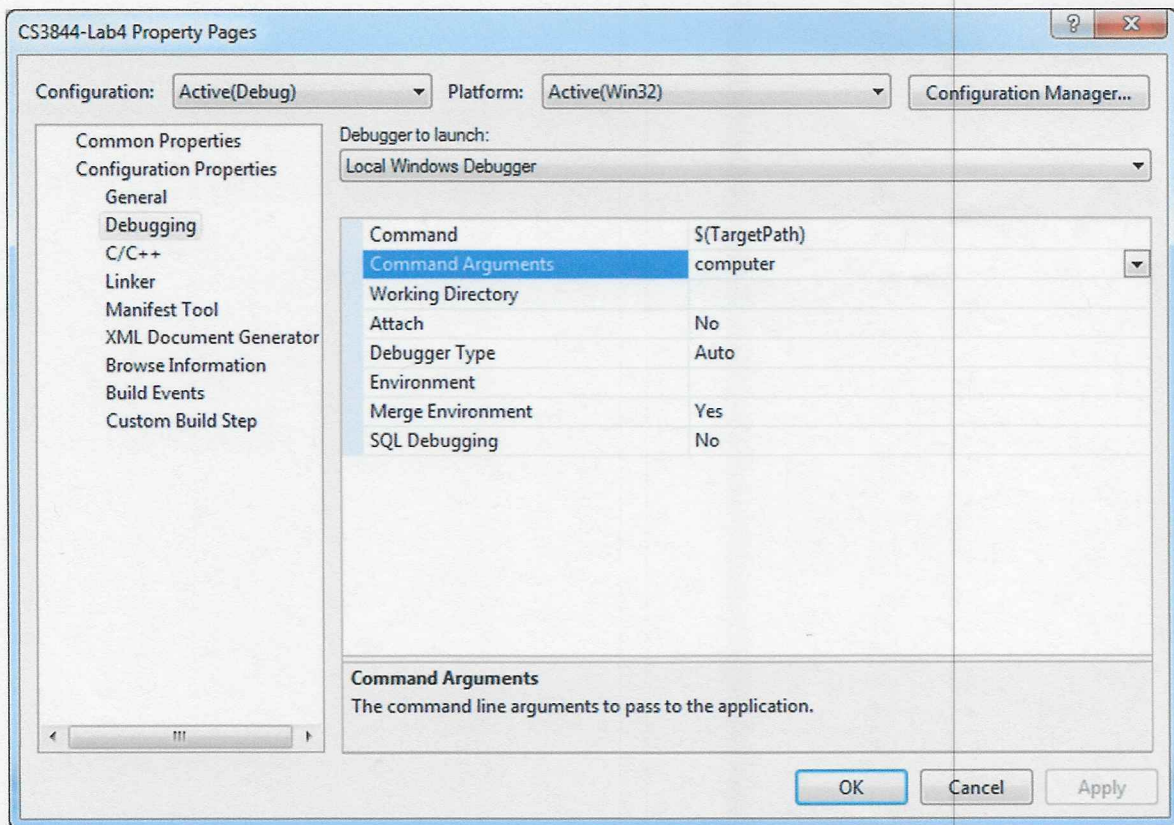


Figure 2 - Set Up Command Argument

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.

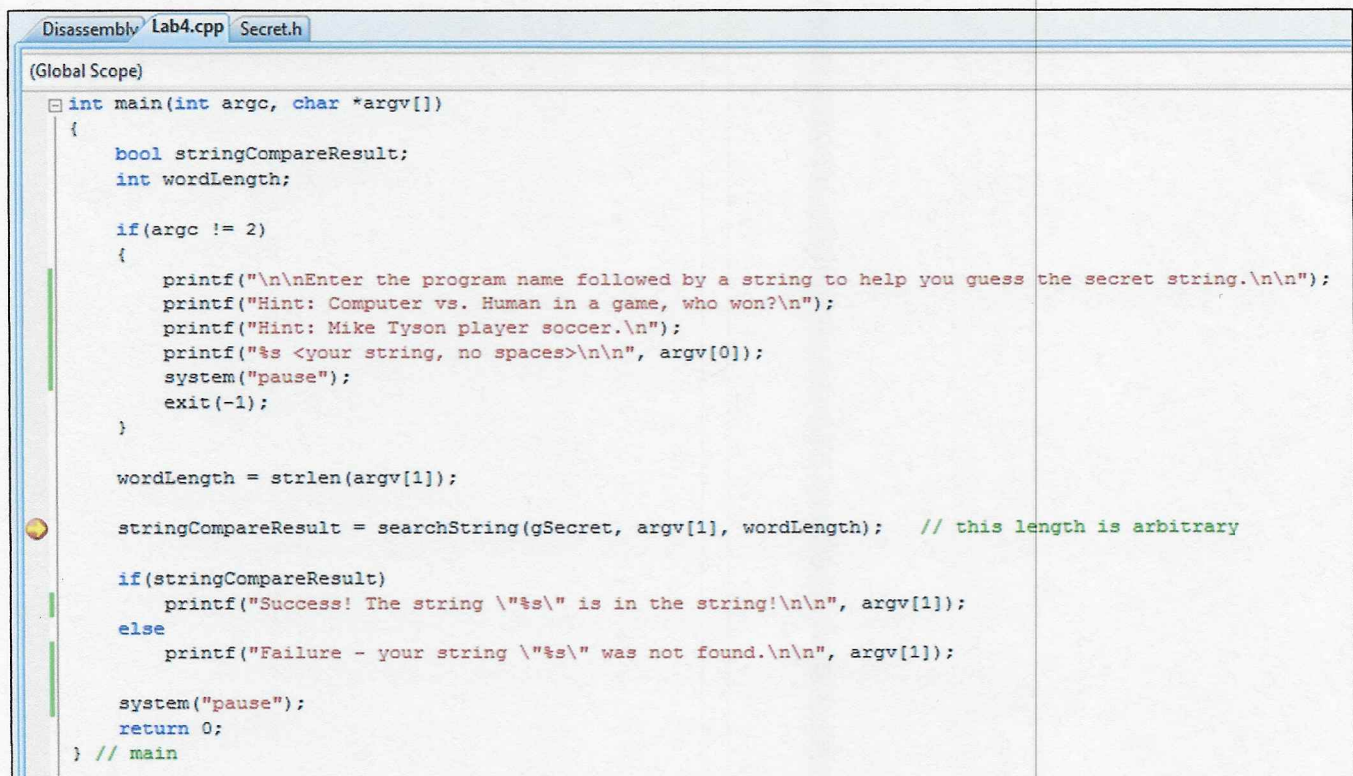


Figure 3 - C Source Code



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

The screenshot displays a debugger's Disassembly window. The top pane shows the disassembled code for a function named `main(int, char**)`. The code includes several instructions: `add esp, 4`, `cmp esi, esp`, `call @ILT+325(__RTC_CheckEsp) (41114Ah)`, `exit(-1);`, `mov esi, esp`, `push 0FFFFFFFh`, `call dword ptr [__imp__exit (4182C4h)]`, `cmp esi, esp`, `call @ILT+325(__RTC_CheckEsp) (41114Ah)`, `wordLength = strlen(argv[1]);`, `mov eax, dword ptr [argv]`, `mov ecx, dword ptr [eax+4]`, `push ecx`, `call @ILT+165(_strlen) (4110AAh)`, `add esp, 4`, `mov dword ptr [wordLength], eax`, `stringCompareResult = searchString(gSecret, argv[1], wordLength);` (commented as // this length is arbitrary), `movzx eax, word ptr [wordLength]`, `push eax`, `mov ecx, dword ptr [argv]`, `mov edx, dword ptr [ecx+4]`, `push edx`, `push offset gSecret (417000h)`, `call searchString (41103Ch)`, `add esp, 0Ch`, `mov byte ptr [stringCompareResult], al`, `if(stringCompareResult)`, `movzx eax, byte ptr [stringCompareResult]`, `test eax, eax`, `je main+108h (411588h)`, `printf("Success! The string \"%s\" is in the string!\n\n", argv[1]);`, `mov esi, esp`, `mov eax, dword ptr [argv]`, `mov ecx, dword ptr [eax+4]`, `push ecx`, `push offset string "Success! The string \"%s\" is in t"... (415770h)`, `call dword ptr [__imp__printf (4182BCh)]`, and `add esp, 8`. The bottom pane shows two memory windows: 'Memory1' with address `esp` and 'Memory2' with address `0x0018FF1C`. The memory contents are displayed in hexadecimal and ASCII.

```

Disassembly Lab4.cpp Secret.h
Address: main(int, char**)
00411517 83 C4 04      add     esp, 4
0041151A 3B F4        cmp     esi, esp
0041151C E8 29 FC FF FF call    @ILT+325(__RTC_CheckEsp) (41114Ah)
        exit(-1);
00411521 8B F4        mov     esi, esp
00411523 6A FF        push    0FFFFFFFh
00411525 FF 15 C4 82 41 00 call    dword ptr [__imp__exit (4182C4h)]
0041152B 3B F4        cmp     esi, esp
0041152D E8 18 FC FF FF call    @ILT+325(__RTC_CheckEsp) (41114Ah)
        }

        wordLength = strlen(argv[1]);
00411532 8B 45 0C      mov     eax, dword ptr [argv]
00411535 8B 48 04      mov     ecx, dword ptr [eax+4]
00411538 51          push    ecx
00411539 E8 6C FB FF FF call    @ILT+165(_strlen) (4110AAh)
0041153E 83 C4 04      add     esp, 4
00411541 89 45 EC      mov     dword ptr [wordLength], eax

        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]
00411564 85 C0        test    eax, eax
00411566 74 20        je      main+108h (411588h)
        printf("Success! The string \"%s\" is in the string!\n\n", argv[1]);
00411568 8B F4        mov     esi, esp
0041156A 8B 45 0C      mov     eax, dword ptr [argv]
0041156D 8B 48 04      mov     ecx, dword ptr [eax+4]
00411570 51          push    ecx
00411571 68 70 57 41 00 push    offset string "Success! The string \"%s\" is in t"... (415770h)
00411576 FF 15 BC 82 41 00 call    dword ptr [__imp__printf (4182BCh)]
0041157C 83 C4 08      add     esp, 8

Memory1
Address: esp
0x0018FE4C 00 00 00 00 00 00 00 00 00 00 e0 fd 7e cc cc cc cc .....â¸¸~ïïïï

Memory2
Address: 0x0018FF1C
0x0018FF1C 08 00 00 00 cc

```

Figure 4 - Disassembled Source Code

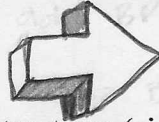


1. Notice that I put ESP in the memory window. From Figure 4 what are EIP and ESP? Your EIP should match if you disabled ASLR but your ESP could vary, which is OK.

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?

MOV EBX, EAX  
CALL BP : EAX = 0029FEF2



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

ESP: 0018FE4C  
mem: 00000000

4810  
-4C  
4

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

mem: 00000008  
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 = 0054FE80

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

mem: 00 00 00 08

00 25 FE F2

gSecret

00 4A E0 00

return addr

00 43 4D 83

add esp, 0C



Disassembly Lab4.cpp Secret.h

Address: searchString(char \*, char \*, unsigned short)

```

//
// This program will be used for Lab#4
//

#include <windows.h>
#include <stdio.h>
#include "Secret.h"

extern char gSecret[];

bool searchString(char *string2Search, char *searchWord, WORD wordLength)
{
004113C0 55          push     ebp
004113C1 8B EC       mov     ebp, esp
004113C3 81 EC CC 00 00 00 sub     esp, 0CCh
004113C9 53          push     ebx
004113CA 56          push     esi
004113CB 57          push     edi
004113CC 8D BD 34 FF FF FF lea     edi, [ebp-0CCh]
004113D2 B9 33 00 00 00 mov     ecx, 33h
004113D7 B8 CC CC CC CC mov     eax, 0CCCCCCCCh
004113DC F3 AB       rep stos dword ptr es:[edi]
    bool result = false;
004113DE C6 45 FB 00 mov     byte ptr [result], 0

    __asm
    {
        xor eax, eax
004113E2 33 C0       xor     eax, eax
        mov esi, searchWord
004113E4 8B 75 0C     mov     esi, dword ptr [searchWord]
        test esi, esi
004113E7 85 F6       test    esi, esi
        je NOT_FOUND
004113E9 74 29       je     NOT_FOUND (411414h)

        mov edi, string2Search
004113EB 8B 7D 08     mov     edi, dword ptr [string2Search]
        test edi, edi
004113EE 85 FF       test    edi, edi
        je NOT_FOUND
004113F0 74 22       je     NOT_FOUND (411414h)
    }
}

```

Memory1

Address: 0x0018FE3C

| Address   | 0x0018FE3C                                      | 0x0018FE4C                                   | 0x0018FE5C                                   | 0x0018FE6C |
|---|---|--|--|------------|
| 5a 15 41 00 00 70 41 00 5a 49 77 00 08 00 00 00 | 00 00 00 00 00 00 00 00 00 e0 fd 7e cc cc cc cc | cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc | cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc |            |

your return address

← 0x0018FE3C

↓ 0x0018FE4C

↓ 0x0018FE5C

↓ 0x0018FE6C

Columns: 16

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 compare, 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]
00411564 0F 00

```

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?

cc cc cc 0cc

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.